2nd Edition
Updated for Python 3



Think Python

HOW TO THINK LIKE A COMPUTER SCIENTIST

Allen B. Downey

目錄

像计算机科学家一样思考python 第二版

作者：Allen B. Downey

翻译：cycleuser

来源：ThinkPython-en-cn

Think Python

> 2nd Edition
>
> by Allen B. Downey
>
> 翻译 大胖哥

This is a very classical book for newbies of programing. There are some versions of translation. But I still think that we can do it by ourselves.

> 这是一本很经典的编程入门书籍。网上有过一些翻译，不过我觉得咱们还是自己来一下比较好！

This is the second edition of Think Python. It uses Python 3, with notes on differences in Python 2. If you are using Python 2, you might want to use the first edition.

> 这是Think Python这本书的第二版，本次使用的是Python3，与Python2有很多不同，这些不同之处会有标注。如果你用Python2的货，还是建议你去阅读上一个版本。

Buy the first edition at Amazon.com

> 到亚马逊购买本书

Download Think Python 2e in PDF.

> 下载 Think Python 2e PDF格式的电子版.

Read Think Python 2e in HTML.

> 在线阅读 Think Python 2e HTML网页版本（推荐这个，都是文字格式，更方便）.

Example programs and solutions to some problems are here (links to specific examples are in the book).

> 样例代码以及其他问题的解决可以到这里找(具体样例的链接在书中就有)。

Description 说明

Think Python is an introduction to Python programming for beginners.

> Think Python 这本书是面向初学者介绍Python编程。

It starts with basic concepts of programming, and is carefully designed to define all terms when they are first used and to develop each new concept in a logical progression.

> 首先介绍的是一些编程的基本内容，给出概念和解释，然后循序渐进地深入讲解每个概念。

Larger pieces, like recursion and object-oriented programming are divided into a sequence of smaller steps and introduced over the course of several chapters.

> 复杂的部分，比如递归以及面向对象编程，这些都分成一个个小块，以多个章节的方式来逐步介绍。

What's new in the second edition?

> 第二版有啥新东西呢？

- We've upgraded to Python 3: All examples in the book are now Python 3, and the supporting code has been updated to run in both Python 2 and 3.

> 开始用Python3了：书里面所有样例都用Python3来实现，参考代码也都做了升级，用Python2或者3都能运行。

- We've removed the roadblocks: Based on reader feedback, we know where people had problems, so we've fixed or removed the pain points.

> 去掉了一些比较难的内容：基于读者反馈，我们认识到大家存在某些困难，所以就调整或者去掉了一些难点。

- Python in the browser: For beginners, one of the challenges of getting started is installing Python. For readers who don't want to install Python right away, we provide instructions for running Python in a browser using PythonAnywhere, a free online programming environment.

> 浏览器内能Python编程了：初学者遇到的第一个困难就是安装Python。另外有的读者可能不想去直接就安装Python，我们就提供了一个用浏览器来运行Python的简介：使用PythonAnywhere，一个免费的在线Python编程环境。（译者注：中国用户以考虑试试fenby.com，也有类似的实现，还有视频的介绍。）

- More Python goodies: We've added a chapter to cover some powerful Python features that didn't make it into the first edition, including list comprehensions and additional data structures.

> 更多Python彩蛋：单独加了一章，来介绍一些第一版中没有提及的Python功能，比如列表解析和附加的数据结构。

Think Python 2e is a Free Book.

> 这本书是一本自-和谐-由的书。

If you have comments, corrections or suggestions, please send me email at feedback{at}thinkpython{dot}com.

> 如果你有一些评论、修正或者建议，可以发邮件给feedback@thinkpython.com

Other Free Books by Allen Downey are available from Green Tea Press.

> 其他由 Allen Downey 编写的自-和谐-由书籍都可以在Green Tea Press找到.

Download 下载

- Precompiled copies of the book are available in PDF.

> 编译好的PDF版本在这里下载：PDF。

- The LaTeX source code is available from this GitHub repository.

> LaTeX代码在GitHub这里可以下载：this GitHub repository.

- Here is the HTML version, and here is the HTML code in a zip archive.

> 网页在线阅读 HTML version, 这些网页打包下载 HTML code，这是一个zip格式的压缩包。

Earlier Versions

> 早期版本

The first edition is here.

> 第一版在这里

A previous edition of this book was published by Cambridge University press with the title Python for Software Design. This edition is available from Amazon.com.

> 本书的上一个版本是由剑桥大学出版社出版的，标题是 Python for Software Design. 可以到亚马逊去买到哈。

The original Python version of the book was published by Green Tea Press with the title How to Think Like a Computer Scientist: Learning with Python. This edition is available from from Lulu.com.

> 本书的原始版本由Green Tea Press 出版，标题为 How to Think Like a Computer Scientist: Learning with Python. 这个版本可以从这个网站找到 Lulu.com.

Other Free Books by Allen Downey are available from Green Tea Press.

> 其他由 Allen Downey 编写的自-和谐-由书籍都可以在Green Tea Press找到.

Chapter 0 Preface 前言

The strange history of this book

> 本书的奇幻历史

In January 1999 I was preparing to teach an introductory programming class in Java. I had taught it three times and I was getting frustrated. The failure rate in the class was too high and, even for students who succeeded, the overall level of achievement was too low.

> 在1999年1月的时候呢，我正准备教一门Java的入门编程课。我当时已经教过三次了，受挫感很强。班上挂科率特别高，而且即使那些没挂科的学生编程的整体水平也特别低。

One of the problems I saw was the books. They were too big, with too much unnecessary detail about Java, and not enough high-level guidance about how to program. And they all suffered from the trap door effect: they would start out easy, proceed gradually, and then somewhere around Chapter 5 the bottom would fall out. The students would get too much new material, too fast, and I would spend the rest of the semester picking up the pieces.

> 当时有很多问题，我首先发现的就是教材。那些教材都特别大部头，有很多关于Java的细节，特别琐碎又并不重要，而且也没有足够的关于如何编程的高层次指导（译者注：就是缺乏战略性指导，没有告诉学生编程的心法）。这些教材总有一些『陷阱门效应』：开头他们都却是挺简单，然后逐步提升，接着突然在某个地方，比如第五章，出现很坑很复杂的陷阱。学生们要突然一下子应对太多新东西，甚至措手不及，而我作为教师就得花费整个后半个学期来一点点给学生们补上。

Two weeks before the first day of classes, I decided to write my own book. My goals were:

> 开课的两周之前，我最终决定要写个自己的教科书。目标如下：

- Keep it short. It is better for students to read 10 pages than not read 50 pages.

> 简短。让学生读10页就够比让他们读50页效果好得多。

- Be careful with vocabulary. I tried to minimize jargon and define each term at first use.

> 降低词汇难度。我尽量把术语用量降到最低，并且在首次使用的时候对每一个都进行定义。

- Build gradually. To avoid trap doors, I took the most difficult topics and split them into a series of small steps.

> 循序渐进。为了避免『陷阱门效应』，我专门把这些最为复杂的部分抽离成一个个专题，并且都切分成小规模的部分，一步步来进行。

- Focus on programming, not the programming language. I included the minimum useful subset of Java and left out the rest.

> 专注于编程，而不是编程语言。我只保留了关于Java的最小规模内容，没有涉及更多的细节。

I needed a title, so on a whim I chose How to Think Like a Computer Scientist.

> 我还需要个标题，就突发奇想，选了个标题叫做『如何像计算机科学家一样思考』。

My first version was rough, but it worked. Students did the reading, and they understood enough that I could spend class time on the hard topics, the interesting topics and (most important) letting the students practice.

> 我的第一版教材很粗糙，但挺管用。学生真能看进去，并且理解了我在课上所讲的那些难点和有趣的专题，最重要的是，他们能够据此来实践。

I released the book under the GNU Free Documentation License, which allows users to copy, modify, and distribute the book.

> 之后我就以GNU自由文档协议来发布了这本书，这一协议允许所有人去复制、修改以及分发这本书。

What happened next is the cool part. Jeff Elkner, a high school teacher in Virginia, adopted my book and translated it into Python. He sent me a copy of his translation, and I had the unusual experience of learning Python by reading my own book. As Green Tea Press, I published the first Python version in 2001.

> 接下来的事情很有趣了。Jeff Elkner，维吉尼亚的一位高中教师，他很欣赏我这本书，把这本书从Java翻译成了Python的版本。他发给我一份『译稿』，然后我开启了阅读『自己的书』来学习Python的奇妙经历。于是在2001年，我通过Green Tea Press出版了本书的第一个Python版本。

In 2003 I started teaching at Olin College and I got to teach Python for the first time. The contrast with Java was striking. Students struggled less, learned more, worked on more interesting projects, and generally had a lot more fun.

> 在2003年，我开始在奥林商学院教学，并且第一次开始教Python了。这和Java的对比很鲜明。学生们省力多了，学得也更多了，在有趣的项目上也更努力，整体上都觉得这一学习过程很有乐趣。

Since then I've continued to develop the book, correcting errors, improving some of the examples and adding material, especially exercises.

> 从那以后，我就继续维护这本书，修正错误，改进样例、附加资料以及练习题。

The result is this book, now with the less grandiose title Think Python. Some of the changes are:

- I added a section about debugging at the end of each chapter. These sections present general techniques for finding and avoiding bugs, and warnings about Python pitfalls.
- I added more exercises, ranging from short tests of understanding to a few substantial projects. Most exercises include a link to my solution.
- I added a series of case studies—longer examples with exercises, solutions, and discussion.
- I expanded the discussion of program development plans and basic design patterns.
- I added appendices about debugging and analysis of algorithms.

> 结果就产生了现在这本书，现在标题简化了很多——Think Python。主要的改变如下：
>
> - 在每一章的末尾，我加了关于debug的部分。这些内容提供了关于debug的一些整体策略，比如如何找到和避免bug，还有就是关于Python一些陷阱进行了提醒。
> - 我加了更多的练习，从简单的理解方面的测试，到一些比较充足的项目。大多数练习都有解决方案的样本链接。
> - 我还添加了一些案例研究，包含练习、解决方案和讨论的更大规模的样例。
> - 此外我还扩展了关于程序开发规划和基本设计模式的讨论。
> - 关于debug和算法分析，还额外加了一些附录。

The second edition of Think Python has these new features:

- The book and all supporting code have been updated to Python 3.
- I added a few sections, and more details on the web, to help beginners get started running Python in a browser, so you don't have to deal with installing Python until you want to.
- For Chapter 4.1 I switched from my own turtle graphics package, called Swampy, to a more standard Python module, turtle, which is easier to install and more powerful.
- I added a new chapter called "The Goodies", which introduces some additional Python features that are not strictly necessary, but sometimes handy.

> 这本Think Python 的第二版有如下的新内容：
>
> - 本书内的所有参考代码都升级到Python3了。
> - 我增加了一部分内容，以及一些关于web方面的细节，这是为了帮助初学者能够在浏览器中开始尝试Python，这样即便你不想安装Python也没问题了。
> - 在第四章的第一节，我把我自己的一个原来叫做Swampy的小乌龟图形包转换撑了一个更标准的Python模块，名字叫做turtle，更好安装，功能也比之前强大了。
> - 我还添加了新的一章，叫做『彩蛋』，介绍了一些Python的额外功能，严格来说，这些功能并不算必备的，但有时候蛮好用的。

I hope you enjoy working with this book, and that it helps you learn to program and think like a computer scientist, at least a little bit.

> 我希望大家能享受学习这本书的过程，也希望这本书能帮助大家学习编程，并且让大家学会像计算机科学家一样思考，哪怕有一点点也好。

Allen B. Downey

> 艾伦 唐尼

Olin College

> 奥林商学院

Acknowledgments

> # 致谢

Many thanks to Jeff Elkner, who translated my Java book into Python, which got this project started and introduced me to what has turned out to be my favorite language.

> 非常感谢Jeff Elkner，是他把我的Java教材翻译成了Python，才引起了这一项目的开始，并且也把Python语言介绍给我，它已经是我最喜欢的编程语言了。

Thanks also to Chris Meyers, who contributed several sections to How to Think Like a Computer Scientist.

> 也要感谢Chris Meyers，他对『如何像计算机科学家一样思考』的一些章节有贡献。

Thanks to the Free Software Foundation for developing the GNU Free Documentation License, which helped make my collaboration with Jeff and Chris possible, and Creative Commons for the license I am using now.

> 感谢自由软件基金会，是他们提出了GNU自由文档协议，在这一协议的帮助下，我和Jeff以及Chris的合作成为了可能，当然也要感谢我现在使用的知识共享协议。

Thanks to the editors at Lulu who worked on How to Think Like a Computer Scientist.

> 感谢Lulu的编辑们，他们出版了『如何像计算机科学家一样思考』。

Thanks to the editors at O'Reilly Media who worked on Think Python.

> 感谢O'Reilly公司的编辑们，他们出版了这本『Think Python』。

Thanks to all the students who worked with earlier versions of this book and all the contributors (listed below) who sent in corrections and suggestions.

> 最后还要感谢所有曾对本书早期版本做出过贡献的同学们，以及其他参与改错和提出建议的朋友们（列表如下）。

Contributor List

> ## 贡献列表

More than 100 sharp-eyed and thoughtful readers have sent in suggestions and corrections over the past few years. Their contributions, and enthusiasm for this project, have been a huge help.

> 百名以上目光敏锐又思维迅捷的读者都在过去的这些年里发来了各种建议或是发现了各种错误。他们贡献和热情都是对本项目的巨大帮助。

If you have a suggestion or correction, please send email tofeedback@thinkpython2.com. If I make a change based on your feedback, I will add you to the contributor list (unless you ask to be omitted).

> 如果大家有任何意见建议，请发邮件到feedback@thinkpython2.com联系我们。如果基于反馈做出了修改，我会将你添加到贡献列表（当然你不想被添加也可以的）。

If you include at least part of the sentence the error appears in, that makes it easy for me to search. Page and section numbers are fine, too, but not quite as easy to work with. Thanks!

> 希望你能至少把出错句子的一部分提供出来，这都让我更容易去搜索。页码和章节编号也可以，但不太容易找。多谢了！

（译者注：以下贡献列表不翻译了）

Chapter 1 The way of the program 编程之路

The goal of this book is to teach you to think like a computer scientist. This way of thinking combines some of the best features of mathematics, engineering, and natural science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assembling components into systems and evaluating tradeoffs among alternatives. Like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.

> 本书的目的是叫你学会像计算机科学家一样来思考。这种思考方式汇聚了数学、工程和自然科学的精华。计算机科学家像数学家一样，使用规范的语言来阐述思想（尤其是一些计算）；像工程师一样设计、组装系统，并且在多重选择中寻找最优解；像自然科学家一样观察复杂系统的行为模式，建立猜想，测试预估的结果。

The single most important skill for a computer scientist is problem solving. Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills. That's why this chapter is called, "The way of the program".

> 计算机科学家唯一最重要的技能就是『解决问题』。解决问题意味着要有能力把问题进行方程化，创造性地考虑解决思路，并且清晰又精确地表达出解决方案。而学习编程的过程，正是一个培养这种解决问题能力的绝佳机会。本章的标题是『编程之路』，原因就在此。

On one level, you will be learning to program, a useful skill by itself. On another level, you will use programming as a means to an end. As we go along, that end will become clearer.

> 在一定层面上，大家将通过编程本身来学习编程这一重要的技巧。在另外一些层面上，大家也将把编程作为实现一种目的的途径。这一目的会随着我们逐渐学习而越发清楚。

1.1 What is a program? 程序是什么？

A program is a sequence of instructions that specifies how to perform a computation. The computation might be something mathematical, such as solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, such as searching and replacing text in a document or something graphical, like processing an image or playing a video.

> 程序是一个指令的序列，来告诉机器如何进行一组运算。这种运算也许是数学上的，比如求解一组等式或者求多项式的根；当然也可以是符号运算，比如在文档中搜索和替换文字，或者一些图形化过程，比如处理图像或者播放一段视频。

The details look different in different languages, but a few basic instructions appear in just about every language:

- input: Get data from the keyboard, a file, the network, or some other device.
- output: Display data on the screen, save it in a file, send it over the network, etc.
- math: Perform basic mathematical operations like addition and multiplication.
- conditional execution: Check for certain conditions and run the appropriate code.
- repetition: Perform some action repeatedly, usually with some variation.

> 不同编程语言的具体细节看着很不一样，但几乎所有编程语言都会有一些基础指令：
>
> - 输入系统：从键盘、文件、网络或者其他设备上获得数据。
> - 输出系统：将数据在屏幕中显示，或者存到文件中、通过网络发送等等。
> - 数学运算：进行基本的数学操作，比如加法或者乘法。
> - 条件判断：检查特定条件是否满足来运行相应的代码。
> - 重复判断：重复进行一些操作，通常会有些变化。

Believe it or not, that's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of instructions that look pretty much like these. So you can think of programming as the process of breaking a large, complex task into smaller and smaller subtasks until the subtasks are simple enough to be performed with one of these basic instructions.

> 大家可能不太相信，核心内容就这么多。你用过的所有程序，无论多么复杂，都是由一些这样的指令组合而成的。因此大家可以把编程的过程理解成一个把庞大复杂任务进行拆分来解决的过程，分解到适合使用上述的基本指令来解决为止。

1.2 Running Python 运行Python

One of the challenges of getting started with Python is that you might have to install Python and related software on your computer. If you are familiar with your operating system, and especially if you are comfortable with the command-line interface, you will have no trouble installing Python. But for beginners, it can be painful to learn about system administration and programming at the same time.

> 新手在刚接触Python的时候遇到的困难之一就是必须在电脑上安装Python和相关的一些软件。如果你熟悉操作系统，并且还很习惯用命令行界面，那安装Python对你来说就没啥问题了。但对初学者来说，要求他们既要了解系统管理又要学习编程，就可能有些困难了。

To avoid that problem, I recommend that you start out running Python in a browser. Later, when you are comfortable with Python, I'll make suggestions for installing Python on your computer.

> 为了避免这种问题，我推荐大家可以在开始的时候用浏览器来体验Python。熟悉了之后，再安装Python到计算机上。

There are a number of web pages you can use to run Python. If you already have a favorite, go ahead and use it. Otherwise I recommend PythonAnywhere. I provide detailed instructions for getting started athttp://tinyurl.com/thinkpython2e.

> 有很多站点提供在线运行Python的功能。如果你已经用过并且有一定经验了，可以选择你喜欢的。我推荐大家可以试试PythonAnywhere，对此的使用介绍可以在下面的链接中找到http://tinyurl.com/thinkpython2e。

There are two versions of Python, called Python 2 and Python 3. They are very similar, so if you learn one, it is easy to switch to the other. In fact, there are only a few differences you will encounter as a beginner. This book is written for Python 3, but I include some notes about Python 2.

> Python现在有两个主要的分支，即Python2和Python3。如果你学过其中的一个，你会发现他们还挺相似的，而且转换起来也不算难。实际上对于初学者来说，他们只有很细微的差别而已。这本书是用Python3写的，但也会对Python2进行注解。

The Python interpreter is a program that reads and executes Python code. Depending on your environment, you might start the interpreter by clicking on an icon, or by typing python on a command line. When it starts, you should see output like this:

> Python的解释器是一个读取并执行Python代码的程序。根据你的系统环境，你可以点击图标或者在命令行中输入python来运行解释器。它运行起来，你会看到类似这样的输出：

```
Python 3.4.0 (default, Jun 19 2015, 14:20:21)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>>
```

The first three lines contain information about the interpreter and the operating system it's running on, so it might be different for you. But you should check that the version number, which is 3.4.0 in this example, begins with 3, which indicates that you are running Python 3.

If it begins with 2, you are running (you guessed it) Python 2.

> 开头的三行包含了关于解释器和所在操作系统的信息，所以大家各自的情况可能有所不同。不过当你检查版本的时候，比如例子中的是3.4.0，使用3开头的，那就告诉你了，他运行的是Python3。你肯定也能猜到，如果开头的是2那就是Python2咯。

The last line is a prompt that indicates that the interpreter is ready for you to enter code. If you type a line of code and hit Enter, the interpreter displays the result:

> 最后一行那个是提示符，告诉你解释器已经就绪了，你可以输入代码了。如果你输入一行代码然后回车键，解释器就会显示结果了，如下所示：

```
>>> 1 + 1
>>> 1 + 1
2
```

Now you're ready to get started. From here on, I assume that you know how to start the Python interpreter and run code.

> 现在你已经做好开始学习Python的准备了。现在我估计你应该已经知道怎么来启动Python解释器和运行Python代码了。

1.3 The first program 第一个程序

Traditionally, the first program you write in a new language is called "Hello, World!" because all it does is display the words "Hello, World!". In Python, it looks like this:

> 传统意义上，大家学一门新编程语言要写的第一个程序都被叫做『Hello，World！』，因为这第一个程序就用来显示这个词组『Hello，World！』。在Python中，是这样实现的：

```
>>> print('Hello, World!')
>>> print('Hello, World!')
```

This is an example of a print statement, although it doesn't't actually print anything on paper. It displays a result on the screen. In this case, the result is the words Hello, World! The quotation marks in the program mark the beginning and end of the text to be displayed; they don't appear in the result.

> 这是一个打印语句的例子，虽然并没有往纸张上面进行实际的『打印』。这个程序把结果显示在屏幕上。结果就是输出了这个词组『Hello，World！』

The parentheses indicate that print is a function. We'll get to functions in Chapter 3.

> 括号表明了print是一个函数。关于函数我们到第三章再讨论。

In Python 2, the print statement is slightly different; it is not a function, so it doesn't use parentheses.

> 在Python2中，打印的语句有一点点不一样：print不是一个函数，所以就不用有括号了。

```
>>> print 'Hello, World!'
>>> print 'Hello, World!'
```

This distinction will make more sense soon, but that's enough to get started. 这个区别以后会理解更深入，现在说这点就够了。

1.4 Arithmetic operators 运算符

After “Hello, World”, the next step is arithmetic. Python provides operators, which are special symbols that represent computations like addition and multiplication.

> 在『Hello，World！』之后，下一步就是运算了。Python提供了运算符，就是一些用来表示例如加法、乘法等运算的符号了。

The operators +, -, and * perform addition, subtraction, and multiplication, as in the following examples:

> 运算符+，-和*表示加法、减法和乘法，如下所示：

```
>>> 40 + 2
>>> 40 + 2
42
>>> 43 - 1
>>> 43 - 1
42
>>> 6 * 7
>>> 6 * 7
42
```

The operator / performs division:

> 运算符右斜杠/意味着除法：

```
>>> 84 / 2
>>> 84 / 2
42.0
```

You might wonder why the result is 42.0 instead of 42. I'll explain in the next section.

> 你估计在纳闷为啥结果是42.0而不是42，这个下一章节我再解释。

Finally, the operator ** performs exponentiation; that is, it raises a number to a power:

> 最后，再说个运算符**，它表示乘方，就是前一个数为底数，后一个数为指数的次幂运算：

```
>>> 6**2 + 6
>>> 6**2 + 6
42
```

In some other languages, ^ is used for exponentiation, but in Python it is a bitwise operator called XOR. If you are not familiar with bitwise operators, the result will surprise you:

> 在其他的一些编程语言中，^这个符号是乘方的意思，但在Python中这是一个位运算操作符叫做『异或』。要是你不熟悉位运算操作符，结果一定让你很惊讶：

```
>>> 6 ^ 2
>>> 6 ^ 2
4
```

I won't cover bitwise operators in this book, but you can read about them at Wiki. 我在本书中不会涉及到位运算，但你可以在下面这个链接里面读一下来了解：Wiki。

1.5 Values and types 值和类型

A value is one of the basic things a program works with, like a letter or a number. Some values we have seen so far are 2, 42.0, and 'Hello, World!'.

> 值就是一个程序操作的基本对象之一，比如一个字母啊，或者数字。刚刚我们看到了一些值的例子了，比如2，42.0，还有那个字符串『Hello，World！』

These values belong to different types: 2 is an integer, 42.0 is a floating-point number, and 'Hello, World!' is a string, so-called because the letters it contains are strung together.

> 这些值属于不同的类型：2是一个整形值，42.0是浮点数，『Hello，World！』是字符串咯。之所以叫字符串就是因为有一串字符。（译者注：这本书的作者真心掰开揉碎地讲解每一个点啊，高中生甚至初中生都应该理解起来没有什么问题，所以大家用这本书来学编程绝对是最佳选择了。）

If you are not sure what type a value has, the interpreter can tell you:

> 如果你不确定一个值是什么类型呢，你可以让解释器来告诉你：

```
>>> type(2)
>>> type(2)
<class 'int'>
>>> type(42.0)
>>> type(42.0)
<class 'float'>
>>> type('Hello, World!')
>>> type('Hello, World!')
<class 'str'>
```

In these results, the word “class” is used in the sense of a category; a type is a category of values.

> 在这些例子中，『class』这个字样表明这是一类，一种类型就是对值的一种划分。

Not surprisingly, integers belong to the type int, strings belong to str and floating-point numbers belong to float.

> 很自然了，整形的就是int了，字符串就是str了，浮点数就是float了。

What about values like '2' and '42.0'? They look like numbers, but they are in quotation marks like strings.

> 那'2' 和 '42.0'这种是啥呢？他们看着像是数字，但带了单引号了。

```
>>> type('2')
>>> type('2')
<class 'str'>
>>> type('42.0')
>>> type('42.0')
<class 'str'>
```

They’re strings.

> 真相就是——他们也是字符串了。

When you type a large integer, you might be tempted to use commas between groups of digits, as in 1,000,000. This is not a legal integer in Python, but it is legal:

> 咱们现在输入一个大的整数，在中间用逗号分隔试试看，比如1，000，000，并不是Python中合乎语法的整形，但也被接受了：

```
>>> 1,000,000
>>> 1,000,000
(1, 0, 0)
```

That’s not what we expected at all! Python interprets 1,000,000 as a comma-separated sequence of integers. We’ll learn more about this kind of sequence later.

> 出乎意料吧，Python把逗号当做了分隔三个整形数字的分隔符了。我们以后再对这种序列进行讨论。

1.6 Formal and natural languages 公式语言和自然语言

Natural languages are the languages people speak, such as English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

> 自然语言就是人说的语言，比如英语、西班牙语、法语，当然包括中文了。他们往往都不是人主动去设计出来的（当然，人会试图去分析语言的规律），自然而然地发生演进。

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly: Programming languages are formal languages that have been designed to express computations.

> 公式语言是人们为了特定用途设计出来的。比如数学的符号就是一种公式语言，特别适合表达数字和符号只见的关系。化学家也用元素符号和化学方程式来表示分子的化学结构。要注意的是：

编程语言是一种用来表达运算的公式语言。 Formal languages tend to have strict syntax rules that govern the structure of statements. For example, in mathematics the statement 3 + 3 = 6 has correct syntax, but 3 + = 3 $ 6 does not. In chemistry H2O is a syntactically correct formula, but 2Zz is not.

> 公式语言有严格的语法规则和对语句结构的要求。比如数学式3+3=6是正确的，而3+=3￥6就不是了。化学上H2O 是正确的化学式，而2Zz 就不是。

Syntax rules come in two flavors, pertaining to tokens and structure. Tokens are the basic elements of the language, such as words, numbers, and chemical elements. One of the problems with 3 += 3 $ 6 is that $ is not a legal token in mathematics (at least as far as I know). Similarly, 2Zz is not legal because there is no element with the abbreviation Zz.

> 语法规则体现在两个方面，代号和结构。 代号是语言的基础元素，比如单词、数字以及化学元素。3 += 3 $ 6这个式子数学上无意义的一个原因就是因为 $ 并不是数学上的符号(至少我所学的数学是没有这个符号的)。类似地， 2Zz 也不对，因为没有一种化学元素的缩写是 Zz.

The second type of syntax rule pertains to the way tokens are combined. The equation 3 += 3 is illegal because even though + and = are legal tokens, you can’t have one right after the other. Similarly, in a chemical formula the subscript comes after the element name, not before.

> 第二个语法规则是代号必须有严格的组合结构。3 += 3这个式子数学上错误就因为虽然这些符号都是数学符号，但不能把加号等号放一起。类似地，化学方程式中要先写元素名字后写个数，而不是反着。

This is @ well-structured Engli$h sentence with invalid t*kens in it. This sentence all valid tokens has, but invalid structure with.

> 这句英语的单词和结构都有错误，大家还是能看懂的哈。（译者注，作者故意这样写，来表明人类的自然语言容错率高。）

When you read a sentence in English or a statement in a formal language, you have to figure out the structure (although in a natural language you do this subconsciously). This process is called parsing.

> 你读一句英语或者公式语言中的语句时候，你必须搞清楚结构（虽然在自然语言中大家潜意识就能搞定了）。这就叫做解译。

Although formal and natural languages have many features in common—tokens, structure, and syntax—there are some differences:

- ambiguity: Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.
- redundancy: In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.
- literalness: Natural languages are full of idiom and metaphor. If I say, “The penny dropped”, there is probably no penny and nothing dropping (this idiom means that someone understood something after a period of confusion). Formal languages mean exactly what they say.

> 虽然公式语言和自然语言有很多共同特征，比如代号、结构、语法这些元素，但差别还是显著的，比如：
>
> - 二义性ambiguity:

自然语言充满二义性，也就是歧义了，人们有时候用上下文线索或者其他信息来帮助处理这种情况。公式语言被设计为尽量不具有二义性，这就意味着一个语句往往只有唯一的一种含义，与上下文无关。

> - 冗余性redundancy:

为了弥补歧义，减少误解，自然语言有很多冗余，结果就是经常有废话。公式语言要精简的多。

> - 文字修辞literalness:

自然语言充满习语和隐喻等。比如我说 “The penny dropped”, 可能并不是字面意思说硬币掉了(这个俚语意思是过了一会终于弄明白了)。公式语言的意思严格精准。 Because we all grow up speaking natural languages, it is sometimes hard to adjust to formal languages. The difference between formal and natural language is like the difference between poetry and prose, but more so:

- Poetry: Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.
- Prose: The literal meaning of words is more important, and the structure contributes more meaning. Prose is more amenable to analysis than poetry but still often ambiguous.
- Programs: The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

> 咱们大家都是说着自然语言长大的，要调节到公式语言有时候挺难的。这两者之间的差别有点像诗词和散文，但差别更大：
>
> - 诗词Poetry:

单词的运用要兼顾词义和押韵，诗的整体要有一定的意境或者情感上的共鸣。双关很常见，并且多是故意的。

> - 散文Prose:

文字意思更重要，结构也有重要作用。相比诗词更好理解，但也有一定的双关语歧义。

> - 程序Programs:

计算机程序的意义必须是无歧义和文采修饰的，能完全用代号和结构的方式进行解析。 Formal languages are more dense than natural languages, so it takes longer to read them. Also, the structure is important, so it is not always best to read from top to bottom, left to right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, the details matter. Small errors in spelling and punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

> 公式语言比自然语言要更加密集，读起来也需要更长时间。公式语言的结构也非常重要，所以从头到尾或者从左到右未必就是最佳方式。大家应该学着动脑来解译程序，分辨代号，解析结构。最后要注意的就是在公式语言中，细节特别特别重要。拼写和符号的小错误对于自然语言来说没什么，但对公式语言来说就能带来大问题。

1.7 Debugging 调试

Programmers make mistakes. For whimsical reasons, programming errors are called bugs and the process of tracking them down is called debugging.

> 程序员也会犯错的。由于很奇妙的原因，程序的错误被叫做bug，调试的过程就叫debug了。（译者注：一个传言是最早的计算机中经常有虫子进去导致短路之类的，清理虫子就成了常规调试的操作，流传至今。。。）

Programming, and especially debugging, sometimes brings out strong emotions. If you are struggling with a difficult bug, you might feel angry, despondent, or embarrassed.

> 编程，尤其是调试的过程，有时候会给人带来强烈的挫败感。面对特别复杂的状况，你可能就感到愤怒、压抑，或者特别难受。

There is evidence that people naturally respond to computers as if they were people. When they work well, we think of them as teammates, and when they are obstinate or rude, we respond to them the same way we respond to rude, obstinate people (Reeves and Nass, The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places).

> 别担心，这些都是正常人对计算机的正常反应。计算机工作正常了，我们会觉得他们像是队友一样；一旦工作出错了，对我们很粗暴，我们对他们的反应就像是对待粗暴可恨的人一样（参考Reeves和Nass，The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places）。

Preparing for these reactions might help you deal with them. One approach is to think of the computer, as an employee with certain strengths, like speed and precision, and particular weaknesses, like lack of empathy and inability to grasp the big picture.

> 为这些反应做好心理准备，这样你在遇到类似情况就更好应对了。我们也可以把计算机当做一个有一定优点但也有特定缺陷的员工，比如速度快精度高，但缺乏共鸣和应对大场面的能力。

Your job is to be a good manager: find ways to take advantage of the strengths and mitigate the weaknesses. And find ways to use your emotions to engage with the problem, without letting your reactions interfere with your ability to work effectively.

> 你的工作就是做个好的经理人：尽量充分利用员工优势并降低他们缺陷的作用。然后想办法把你的情绪用在解决问题上，而不要让过激的反应干扰工作效率。

Learning to debug can be frustrating, but it is a valuable skill that is useful for many activities beyond programming. At the end of each chapter there is a section, like this one, with my suggestions for debugging. I hope they help!

> 调试的过程挺烦人的，但这个本领很有价值，而且在编程之外的其他领域都有用武之地。在每一章的末尾，都会有这样的一段，我会给出一些关于调试方面的建议。希望能帮到大家！

1.8 Glossary 术语列表

problem solving: The process of formulating a problem, finding a solution, and expressing it.

> 问题解决：将问题方程化，找到解决方案，并表达出来的过程。

high-level language: A programming language like Python that is designed to be easy for humans to read and write.

> 高级语言：例如Python这样的编程语言，设计初衷为易于被人阅读和书写。

l ow-level language: A programming language that is designed to be easy for a computer to run; also called “machine language” or “assembly language”.

> 低级语言：设计初衷为易于被计算机运行的语言，比如机器语言和汇编语言。

portability: A property of a program that can run on more than one kind of computer.

> 可移植性：程序能运行于多种平台的特性。

interpreter: A program that reads another program and executes it

> 解释器：一边读取一边执行代码的程序。

prompt: Characters displayed by the interpreter to indicate that it is ready to take input from the user.

> 提示符：解释器显示的，提醒用户准备就绪，随时可以输入。

program: A set of instructions that specifies a computation.

> 程序：进行一种特定运算的一系列指令。

print statement: An instruction that causes the Python interpreter to display a value on the screen.

> 打印语句：让Python解释器输出值到屏幕的指令。

operator: A special symbol that represents a simple computation like addition, multiplication, or string concatenation.

> 运算符（操作符）：一系列特殊的符号，表示一些简单的运算，比如加减乘除或者字符串操作。

value: One of the basic units of data, like a number or string, that a program manipulates.

> 值：数据的基本组成单元，比如数字或者字符串，是程序处理的对象。

type: A category of values. The types we have seen so far are integers (typeint), floating-point numbers (type float), and strings (type str).

> 类型：对值的分类，大家刚刚接触到的有整形int，浮点数float，以及字符串str。

integer: A type that represents whole numbers. 整形：就是整数咯。 floating-point: A type that represents numbers with fractional parts.

> 浮点数：简单说，就是有小数点的数了。

string: A type that represents sequences of characters.

> 字符串：一串有序的字符了。

natural language: Any one of the languages that people speak that evolved naturally.

> 自然语言：人们说的语言，自然地演化。

formal language: Any one of the languages that people have designed for specific purposes, such as representing mathematical ideas or computer programs; all programming languages are formal languages.

> 公式语言：人为设计的用于特定用途的语言，比如数学用途或者计算机编程用的；所有编程语言都是公式语言。

token: One of the basic elements of the syntactic structure of a program, analogous to a word in a natural language.

> 代号：程序结构中的一种基本元素，相当于自然语言中的单词。

syntax: The rules that govern the structure of a program.

> 语法：程序语言结构的规则。

parse: To examine a program and analyze the syntactic structure.

> 解译：理解程序并分析语法结构的过程。

bug: An error in a program.

> Bug：程序的错误。

debugging: The process of finding and correcting bugs.

> 调试（debug）：搜索和改正程序错误的过程。

1.9 Exercises 练习

Exercise 1 练习1

It is a good idea to read this book in front of a computer so you can try out the examples as you go.

> 你读这本书的同时最好手边有台电脑，这样你就能把样例在电脑上随时运行来看看效果了。

Whenever you are experimenting with a new feature, you should try to make mistakes. For example, in the “Hello, world!” program, what happens if you leave out one of the quotation marks? What if you leave out both? What if you spell print wrong?

> 无论你学任何一种新功能的时候，都可以试着犯点错误。比如就在这个『Hello，World！』程序，你可以试试去掉一个引号会怎么样，都去掉会怎么样，print这个单词拼错了会怎么样等等。

This kind of experiment helps you remember what you read; it also helps when you are programming, because you get to know what the error messages mean. It is better to make mistakes now and on purpose than later and accidentally.

> 这种尝试能让你对读到的内容有更深刻的记忆；也有助于你编程，因为你在编程的时候也得知道调试信息的意思。所以最好现在就故意犯些错误来看看，比以后毫无准备地遇到要好多了。

1. In a print statement, what happens if you leave out one of the parentheses, or both?

> 在print语句后面的括号去掉一个或者两个，看看会怎么样？

1. If you are trying to print a string, what happens if you leave out one of the quotation marks, or both?

> Print字符串的时候如果你丢掉一个引号或者两个引号试试看会如何？

1. You can use a minus sign to make a negative number like -2. What happens if you put a plus sign before a number? What about 2++2?

> 输入一个负数试试，比如-2。然后再试试在数字前面添加加号会怎么样？比如2++2。

1. In math notation, leading zeros are ok, as in 02. What happens if you try this in Python?

> 数学上计数用零开头是可以得，比如02，在Python下面试试会怎样？

1. What happens if you have two values with no operator between them?

> 两个值中间没有运算符会怎么样？

Chapter 2 Variables, expressions and statements 变量，表达式，语句

One of the most powerful features of a programming language is the ability to manipulate variables. A variable is a name that refers to a value.

> 编程语言最强大的功能就是操作变量。变量就是一个有值的代号。

2.1 Assignment statements 赋值语句

An assignment statement creates a new variable and gives it a value:

> 赋值语句的作用是创建一个新的变量，并且赋值给这个变量：

```
>>> message = 'And now for something completely different'
>>> message = 'And now for something completely different'
>>> n = 17
>>> n = 17
>>> pi = 3.141592653589793
>>> pi = 3.141592653589793
```

This example makes three assignments. The first assigns a string to a new variable named message; the second gives the integer 17 to n; the third assigns the (approximate) value of π to pi.

> 上面就是三个赋值语句的例子。第一个是把一个字符串复制给名叫message的新变量；第二个将n赋值为整数17；第三个把圆周率的一个近似值赋给了pi这个变量。

A common way to represent variables on paper is to write the name with an arrow pointing to its value. This kind of figure is called a state diagram because it shows what state each of the variables is in (think of it as the variable's state of mind). Figure 2.1 shows the result of the previous example.

> 平常大家在纸上对变量赋值的方法就是写下名字，然后一个箭头指向它的值。这种图解叫做状态图，因为它能指明各个变量存储的是什么内容。下图就展示了上面例子中赋值语句的结果。



Figure 2.1: State diagram.

2.2 Variable names 变量名称

Programmers generally choose names for their variables that are meaningful—they document what the variable is used for.

> 编程的人总得给变量起个有一定意义的名字才能记得住，一般情况就用名字来表示这个变量的用途了。

Variable names can be as long as you like. They can contain both letters and numbers, but they can't begin with a number. It is legal to use uppercase letters, but it is conventional to use only lower case for variables names.

> 变量名称你随便起多长都可以的。包含字母或者数字都行，但是不能用数字来开头。大写字母也能用，不过还是建议都用小写字母来给变量命名，这个比较传统哈。

The underscore character, _, can appear in a name. It is often used in names with multiple words, such as your_name or airspeed_of_unladen_swallow.

> 变量名里面可以有下划线_，一般在多个单词组成的变量名里面往往用到下划线，比如your_name等等。

If you give a variable an illegal name, you get a syntax error:

> 你要是给变量起名不合规则，就会出现语法错误提示了：

```
>>> 76trombones = 'big parade'
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

76trombones is illegal because it begins with a number. more@ is illegal because it contains an illegal character, @. But what's wrong with class?

> 第一个数字开头所以不合规则，第二个有非法字符@，第三个这个class咋不行呢？好奇吧？

It turns out that class is one of Python's keywords. The interpreter uses keywords to recognize the structure of the program, and they cannot be used as variable names.

> 因为clas是Python里面的一个关键词啦。解释器要用关键词来识别程序的结构，这些关键词是不能用来做变量名的。

Python 3 has these keywords:

> 以下是Python3的关键词哈：

- False class finally is
- return None continue for lambda
- try True def from nonlocal
- while and del global not
- with as elif if or
- yield assert else import pass
- break except in raise You don’t have to memorize this list. In most development environments, keywords are displayed in a different color; if you try to use one as a variable name, you’ll know.

> 你不用去记忆这些哈。因为一般大多数的开发环境里面，关键词都会有区别于普通代码的颜色提示你，你要是用他们做变量名了，一看就会知道的。

2.3 Expressions and statements 表达式和语句

An expression is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions:

> 表达式是数值,变量和操作符的组合。单个值本身也被当作一个表达式，变量也是如此，下面这些例子都是一些正确表达式：

```
>>> 42
>>> 42
42
>>> n
>>> n
17
>>> n + 25
>>> n + 25
42
```

When you type an expression at the prompt, the interpreter evaluates it, which means that it finds the value of the expression. In this example, n has the value 17 and n + 25 has the value 42.

> 当你在提示符后面敲出一个表达式，解释器就判断一下，他会找到这个表达式的值。在本节的例子中，n的值是17，所以n+25就是42了。

A statement is a unit of code that has an effect, like creating a variable or displaying a value.

> 语句是一组具有某些效果的代码，比如创建变量，或者显示值。

```
>>> n = 17
>>> n = 17
>>> print(n)
>>> print(n)
```

The first line is an assignment statement that gives a value to n. The second line is a print statement that displays the value of n.

> 上面第一个就是赋值语句，给n赋值。第二行是显示n的值。

When you type a statement, the interpreter executes it, which means that it does whatever the statement says. In general, statements don't have values.

> 输入语句的时候，解释器会执行它，就是会按照语句所说的去做。一般语句是没有值的。

2.4 Script mode 脚本模式

So far we have run Python in interactive mode, which means that you interact directly with the interpreter. Interactive mode is a good way to get started, but if you are working with more than a few lines of code, it can be clumsy.

> 以上我们一直在用Python的交互模式，就是直接咱们人跟解释器来交互。开始学的时候这样挺好的，但如果你要想一次运行多行代码，这样就很不方便了。

The alternative is to save code in a file called a script and then run the interpreter in script mode to execute the script. By convention, Python scripts have names that end with .py.

> 所以就有另一种选择了，把代码保存成脚本，然后用脚本模式让解释器来运行这些脚本。通常Python脚本文件的扩展名是.py。

If you know how to create and run a script on your computer, you are ready to go. Otherwise I recommend using PythonAnywhere again. I have posted instructions for running in script mode at http://tinyurl.com/thinkpython2e.

> 如果你知道怎么创建和运行脚本，那就尽管在自己电脑上尝试好了。否则我就建议你还是用PythonAnywhere。关于脚本模式的介绍我放到网上了，打开这个链接（http://tinyurl.com/thinkpython2e）去看下哈。

Because Python provides both modes, you can test bits of code in interactive mode before you put them in a script. But there are differences between interactive mode and script mode that can be confusing.

> Python两种模式都支持，所以你可以先用交互模式做点测试，然后再写成脚本。但是两种模式之间有些区别的，所以可能也挺麻烦。

For example, if you are using Python as a calculator, you might type： 举个例子哈，比如咱们把Python当计算器用，你输入以下内容：

```
>>> miles = 26.2
>>> miles = 26.2
>>> miles * 1.61
>>> miles * 1.61
42.182
```

The first line assigns a value to miles, but it has no visible effect. The second line is an expression, so the interpreter evaluates it and displays the result. It turns out that a marathon is about 42 kilometers.

> 第一行给miles这个变量赋初值（译者注：26.2英里是马拉松比赛全程长度），但是看着没啥效果。第二行是一个表达式，解释器会计算这个表达式，然后把结果输出。结果就是把马拉松全程长度从英里换算成公里，答案是42公里哈。

But if you type the same code into a script and run it, you get no output at all. In script mode an expression, all by itself, has no visible effect. Python actually evaluates the expression, but it doesn't display the value unless you tell it to:

> 不过你要是直接把这些代码存成脚本然后运行，是啥都看不到的，没有输出。在脚本模式表达式是没有明显效果的。Python确实会计算这些表达式，但不显示结果，想看到结果你就得告诉他输出一下：

```
miles = 26.2
print(miles * 1.61)
```

This behavior can be confusing at first.

> 这种情况开始还挺让人混乱的。

A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute.

> 脚本一般都是包含了一系列的语句。如果语句超过一条，每个语句执行的时候都会显示结果。比如下面这个：

For example, the script

```
print(1)
x = 2
print(x)
```

produces the output

> 输出的结果如下

```
1
2
```

The assignment statement produces no output.

> 赋值语句是不会有任何输出的。

To check your understanding, type the following statements in the Python interpreter and see what they do:

> 检查下你理解了没哈，把下面这些语句输入到Python解释器，看看会发生什么：

```
5 x = 5 x + 1
```

Now put the same statements in a script and run it. What is the output? Modify the script by transforming each expression into a print statement and then run it again.

> 现在再把同样的语句输入到脚本中，然后用Python来运行一下。看看输出是啥样的？把脚本中的表达式修改一下，每一个都加一个打印语句再试试。

2.5 Order of operations 运算符优先级

When an expression contains more than one operator, the order of evaluation depends on the order of operations. For mathematical operators, Python follows mathematical convention. The acronym PEMDAS is a useful way to remember the rules:

> 表达式可能会包含不止一个运算符，这些不同的运算先后次序就是运算符的优先级。对于数学运算符来说，Python就遵循着数学上的规则。下面这个PEMDAS、是用来记忆这些优先规则的好方法：

- Parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, 2 *(3-1) is 4, and (1+1)**(5-2) is 8. You can also use parentheses to make an expression easier to read, as in(minute* 100) / 60, even if it doesn't change the result.

> 括号内的内容最优先，大家可以用括号来强制某系表达式有限计算。所以2（*3-1*）就等于*4*了，（*1+1*）**（5-2）就是2的立方，等于*8*。使用括号也有助于让你的表达式读起来更好理解，比如*(minute* 100) / 60，这个也不影响计算结果，不过看起来易于理解。

- Exponentiation has the next highest precedence, so 1 + 2**3 is 9, not 27, and 2 * 3**2 is 18, not 36.

> 除了括号，所有运算符中，乘方最优先，所以1 + 2**3的结果是9而不是27，2*3**2结果是18，而不是36。

- Multiplication and Division have higher precedence than Addition andSubtraction. So 2*3-1 is 5, not 4, and 6+4/2 is 8, not 5.

> 乘除运算比加减优先，译者认为大家都知道了，这个我就不细说了。

- Operators with the same precedence are evaluated from left to right (except exponentiation). So in the expression degrees / 2 * pi, the division happens first and the result is multiplied by pi. To divide by 2 π, you can use parentheses or write degrees / 2 / pi.

> 同类运算符从左往右来进行，乘方除外。这个也不细说了，很简单。

I don’t work very hard to remember the precedence of operators. If I can’t tell by looking at the expression, I use parentheses to make it obvious.

> 我不会花很大力气来记忆这些运算符的优先级。如果我记不起来，就用括号来让优先级明确一些就好了。

2.6 String operations 字符串操作

In general, you can’t perform mathematical operations on strings, even if the strings look like numbers, so the following are illegal:

> 一般情况下，咱们不能对字符串进行数学运算的，即使字符串看上去像是数字也不行，所以以下这些都是非法操作：

```
'2'-'1'    'eggs'/'easy'    'third'*'a charm'
```

But there are two exceptions, + and *.

> 不过+和*可以用在字符串上面。

The + operator performs string concatenation, which means it joins the strings by linking them end-to-end. For example:

> +加号的意思就是字符串拼接了，会把两个字符串拼到一起，如下所示：

```
>>> first = 'throat'
>>> first = 'throat'
>>> second = 'warbler'
>>> second = 'warbler'
>>> first + second
>>> first + second
throatwarbler
```

The *operator also works on strings; it performs repetition. For example,'Spam'*3 is 'SpamSpamSpam'. If one of the values is a string, the other has to be an integer.

> 星号也就是乘法运算符也可以用在字符串上面，效果就是重复。比如'Spam'3 结果就是

'SpamSpamSpam'，重复了三次。需要注意的是字符串必须用整数去乘。 This use of + and *makes sense by analogy with addition and multiplication. Just as 4*3 is equivalent to 4+4+4, we expect 'Spam'*3 to be the same as'Spam'+'Spam'+'Spam', and it is. On the other hand, there is a significant way in which string concatenation and repetition are different from integer addition and multiplication. Can you think of a property that addition has that string concatenation does not?

> 这种加法和乘法实际上就是拼接和重复的意思。就像4*3等同于*4+4+4一样，'Spam'*3 也就等于'Spam'+'Spam'+'Spam'。另外一方面，字符串的拼接重复与整数的加法乘法也有显著区别。你能想到加法具有些特性是字符拼接所不具有的么？

2.7 Comments 注释

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why.

> 程序会越来越庞大，也越复杂了，读起来就会更难了。公式语言很密集，靠阅读来理解代码，总是很困难的。

For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called comments, and they start with the # symbol:

> 为了解决阅读的困难，咱们就可以添加一些笔记到代码中，把程序的功能用自然语言来解释一下。这种笔记就叫注释了，使用井号#来开头的：

```
# compute the percentage of the hour that has elapsed percentage = (minute * 100) / 60
```

In this case, the comment appears on a line by itself. You can also put comments at the end of a line:

> 注释可以另起一行，也可以放到行末尾：

```
percentage = (minute * 100) / 60     # percentage of an hour
```

Everything from the # to the end of the line is ignored—it has no effect on the execution of the program

> 井号#后面的内容都会被忽略，因此不会影响程序的运行结果。

Comments are most useful when they document non-obvious features of the code. It is reasonable to assume that the reader can figure out what the code does; it is more useful to explain why.

> 一般注释都是用来解释代码的一些不明显的特性。一般情况下读代码的人应该能理解代码的功能是什么，所以用注释多是要解释这样做的目的是什么。

This comment is redundant with the code and useless:

> 这个注释就显然是多余的，根本没必要：

```
v = 5     # assign 5 to v
```

This comment contains useful information that is not in the code:

> 这种注释包含了重要信息，就很重要了：

```
v = 5     # velocity in meters/second.
```

Good variable names can reduce the need for comments, but long names can make complex expressions hard to read, so there is a tradeoff.

> 变量命名得当的话，就没必要用太多注释了，不过名字要是太长了，表达式读起来也挺麻烦，所以就得权衡着来了。

2.8 Debugging 调试

Three kinds of errors can occur in a program: syntax errors, runtime errors, and semantic errors. It is useful to distinguish between them in order to track them down more quickly.

> 程序一般会有三种错误：语法错误，运行错误和语义错误。区分这三种错误有助于更快速地追踪错误。

- Syntax error: “Syntax” refers to the structure of a program and the rules about that structure. For example, parentheses have to come in matching pairs, so(1 + 2) is legal, but 8) is a syntax error. If there is a syntax error anywhere in your program, Python displays an error message and quits, and you will not be able to run the program. During the first few weeks of your programming career, you might spend a lot of time tracking down syntax errors. As you gain experience, you will make fewer errors and find them faster.

> 语法错误Syntax error:

语法是指程序的结构和规则。比如括号要成对用。如果你的程序有某个地方出现了语法错误，Python会显示出错信息并退出，程序就不能运行了。最开始学习编程的这段时间，你遇到的最常见的估计就是这种情况。等你经验多了，基本就犯的少了，而且也很容易发现了。

- Runtime error: The second type of error is a runtime error, so called because the error does not appear until after the program has started running. These errors are also called exceptions because they usually indicate that something exceptional (and bad) has happened. Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one.

> 运行错误Runtime error:

第二种错误就是运行错误，显而易见了，就是直到运行的时候才会出现的错误。这种错误也被叫做异常，因为一般表示一些意外的尤其是比较糟糕的情况发生了。

- Semantic error: The third type of error is “semantic”, which means related to meaning. If there is a semantic error in your program, it will run without generating error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

> 语义错误Semantic error:

第三种就是语义错误，顾名思义，是跟意义相关。这种错误是指你的程序运行没问题，也不产生错误信息，但不能正确工作。程序可能做一些和设计目的不同的事情。发现语义错误特别不容易，需要你仔细回顾代码和程序输出，要搞清楚到底程序做了什么。

2.9 Glossary 术语列表

variable: A name that refers to a value. 变量：有值的量。

> assignment:

A statement that assigns a value to a variable.

> 赋值：给一个变量赋予值。

state diagram: A graphical representation of a set of variables and the values they refer to.

> 状态图：图形化表征每个变量的值。

keyword: A reserved word that is used to parse a program; you cannot use keywords like if, def, and while as variable names.

> 关键词：系统保留的用于解析程序的词，不能用关键词当做变量名。

operand: One of the values on which an operator operates.

> 运算数：运算符来进行运算操作的数值。

expression: A combination of variables, operators, and values that represents a single result.

> 表达式：一组变量、运算数的组合，会产生单值作为结果。

evaluate: To simplify an expression by performing the operations in order to yield a single value.

> 求解：把表达式所表示的运算计算出来，得到一个单独的值。

statement: A section of code that represents a command or action. So far, the statements we have seen are assignments and print statements.

> 声明：一组表示一种命令或者动作的代码，目前我们了解的只有赋值语句和打印语句。

execute: To run a statement and do what it says.

> 运行：将一条语句进行运行。

interactive mode: A way of using the Python interpreter by typing code at the prompt.

> 交互模式：在提示符后输入代码，让解释器来运行代码的模式。

script mode: A way of using the Python interpreter to read code from a script and run it.

> 脚本模式：将代码保存成脚本文件，用解释器运行的模式。

script: A program stored in a file.

> 脚本：程序以文本形式存成的文件。

order of operations: Rules governing the order in which expressions involving multiple operators and operands are evaluated.

> 运算符优先级：不同运算符和运算数进行计算的优先顺序。

concatenate: To join two operands end-to-end.

> 拼接：把两个运算对象相互连接到一起。

comment: Information in a program that is meant for other programmers (or anyone reading the source code) and has no effect on the execution of the program.

> 注释：程序中用来解释代码含义和运行效果的备注信息，通常给阅读代码的人准备的。

syntax error: An error in a program that makes it impossible to parse (and therefore impossible to interpret).

> 语法错误：程序语法上的错误，导致程序不能被解释器解译，就不能运行了。

exception: An error that is detected while the program is running.

> 异常：程序运行的时候被探测到的错误。

semantics: The meaning of a program.

> 语义：程序的意义。

semantic error: An error in a program that makes it do something other than what the programmer intended.

> 语义错误：程序运行的结果和料想的不一样，没有完成设计的功能，而是干了点其他的事情。

2.10 Exercises 练习

Exercise 1 练习1

Repeating my advice from the previous chapter, whenever you learn a new feature, you should try it out in interactive mode and make errors on purpose to see what goes wrong.

> 像上一章一样，按我建议的，不论学了什么新内容，你都试着在交互模式上故意犯点错误，看看会怎么样。

- We've seen that n = 42 is legal. What about 42 = n?

> 我们都看到了n=42是可以的，那42=n怎么样？

- How about x = y = 1?

> 再试试x=y=1呢？

- In some languages every statement ends with a semi-colon, ;. What happens if you put a semi-colon at the end of a Python statement?

> 有的语言每个语句结尾都必须有个单引号或者分号，试试在Python句末放个会咋样？

- What if you put a period at the end of a statement?

> 句尾放个句号试试呢？

- In math notation you can multiply x and y like this: x y. What happens if you try that in Python?

> 数学上你可以把x和y相乘写成xy，Python里面你这么试试看？

Exercise 2

Practice using the Python interpreter as a calculator:

1. The volume of a sphere with radius r is 4/3 π r3. What is the volume of a sphere with radius 5?
2. Suppose the cover price of a book is $24.95, but bookstores get a 40% discount. Shipping costs $3 for the first copy and 75 cents for each additional copy. What is the total wholesale cost for 60 copies?
3. If I leave my house at 6:52 am and run 1 mile at an easy pace (8:15 per mile), then 3 miles at tempo (7:12 per mile) and 1 mile at easy pace again, what time do I get home for breakfast?

> 把Python解释器当做计算器来做下面的练习：

1. 球体体积是三分之四倍的圆周率乘以半径立方，求半径为5的球体体积。
2. 加入一本书的封面标价是24.95美元，书店打六折。第一本运费花费3美元，后续每增加一本的运费是75美分。问买60本一共得花多少钱呢？
3. 我早上六点五十二分出门离家，以8:15的节奏跑了一英里，又以7：12的节奏跑了三英里，然后又是8：15的节奏跑一英里，回到家吃饭是几点？

Chapter 3 Functions 函数

In the context of programming, a function is a named sequence of statements that performs a computation. When you define a function, you specify the name and the sequence of statements. Later, you can “call” the function by name.

> 在编程的语境下，“函数”这个词的意思是对一系列语句的组合，这些语句共同完成一种运算。定义函数的时候，你要给这个函数指定一个名字，另外还好写出这些进行运算的语句。定义完成后，就可以通过函数名来“调用”函数。

3.1 Function calls 函数调用

We have already seen one example of a function call:

> 此前我们已经见识过函数调用的一个例子了：

```
>>> type(42)
>>> type(42)
<class 'int'>
```

The name of the function is type. The expression in parentheses is called the argument of the function. The result, for this function, is the type of the argument.

> 这个函数的名字就是tpye，括号里面的表达式叫做函数的参数。这个函数的结果是返回参数的类型。

It is common to say that a function “takes” an argument and “returns” a result. The result is also called the return value. Python provides functions that convert values from one type to another. The int function takes any value and converts it to an integer, if it can, or complains otherwise:

> 一般来说，函数都要“传入”一个参数，“返回”一个结果。结果也被叫做返回值。Python提供了一些转换数值类型的函数。比如int这个函数就可以把值转换成整形，但不是什么都能转的，遇到不能转换的就会报错了，如下所示：

```
>>> int('32')
>>> int('32')
32
>>> int('Hello')
>>> int('Hello')
ValueError: invalid literal for int(): Hello
```

int can convert floating-point values to integers, but it doesn’t round off; it chops off the fraction part:

> int这个函数能把浮点数转成整形，但不是很完美，小数部分就都给砍掉了。

```
>>> int(3.99999)
>>> int(3.99999)
3
>>> int(-2.3)
>>> int(-2.3)
-2
```

float converts integers and strings to floating-point numbers:

> float能把整形和字符串转变成浮点数：

```
>>> float(32)
>>> float(32)
32.0
>>> float('3.14159')
>>> float('3.14159')
3.14159
```

Finally, str converts its argument to a string:

> 最后来看下，str可以把参数转变成字符串：

```
>>> str(32)
>>> str(32)
'32'
>>> str(3.14159)
>>> str(3.14159)
'3.14159'
```

3.2 Math functions

Python has a math module that provides most of the familiar mathematical functions. A module is a file that contains a collection of related functions.

> Python内置了一个数学模块，这一模块提供了绝大部分常用的数学函数。模块就是一系列相关函数的集合成的文件。

Before we can use the functions in a module, we have to import it with an import statement:

> 在使用模块中的函数之前，必须先要导入这个模块，使用导入语句：

```
>>> import math
>>> import math
```

This statement creates a module object named math. If you display the module object, you get some information about it:

> 这个语句建立了一个模块对象，名字叫做math。如果你让这个模块对象显示一下，你就会得到与之相关的信息了：

```
>>> math
>>> math
<module 'math' (built-in)>
```

The module object contains the functions and variables defined in the module. To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period). This format is called dot notation.

> 模块对象包含了一些已经定义好的函数和变量。指定模块名和函数名，要用点（也就是英文的句号）来连接模块名和函数名，就可以调用指定的函数了。

```
>>> ratio = signal_power / noise_power
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)
>>> decibels = 10 * math.log10(ratio)
>>> radians = 0.7
>>> radians = 0.7
>>> height = math.sin(radians)
>>> height = math.sin(radians)
```

The first example uses math.log10 to compute a signal-to-noise ratio in decibels (assuming that signal_power and noise_power are defined). The math module also provides log, which computes logarithms base e. 第一个例子用了数学的log10的函数，来计算信噪比的分贝值（假设信号强度和噪音强度都已知了）。数学模块同事也提供了log，用自然底数e取对数的函数。 The second example finds the sine of radians. The name of the variable is a hint that sin and the other trigonometric functions (cos, tan, etc.) take arguments in radians. To convert from degrees to radians, divide by 180 and multiply by π:

> 第二个例子是对弧度值计算正弦值。通过变量名你应该能推测出正弦以及其他的三角函数（比如余弦、正切等等）都要用弧度值作为参数。所以要把角度的值从度转换成弧度，方法就是除以180然后再乘以圆周率π：

```
>>> degrees = 45
>>> degrees = 45
>>> radians = degrees / 180.0 * math.pi
>>> radians = degrees / 180.0 * math.pi
>>> math.sin(radians)
>>> math.sin(radians)
0.707106781187
```

The expression math.pi gets the variable pi from the math module. Its value is a floating-point approximation of π, accurate to about 15 digits.

> math.pi这个表达式从数学模块中得到π的一个大概精确到15位的近似值，存成一个浮点数。

If you know trigonometry, you can check the previous result by comparing it to the square root of two divided by two:

> 了解了三角函数之后，你可以用试着把2的平方根除以二，然后对比一下这个结果和上一个结果：

```
>>> math.sqrt(2) / 2.0
>>> math.sqrt(2) / 2.0
0.707106781187
```

> 译者注：画一个三角形就知道了，45度角两直角边是单位1，斜边必然是2的平方根了，对应的正弦余弦也都是这个值了。大家应该能理解吧？

3.3 Composition 组合

So far, we have looked at the elements of a program—variables, expressions, and statements—in isolation, without talking about how to combine them.

> 目前为止，我们已经见识了一个程序所需要的大部分元素了：变量、表达式、语句。不过咱们都是一个个单独看到的，还没有把它们结合起来试试。

One of the most useful features of programming languages is their ability to take small building blocks and compose them. For example, the argument of a function can be any kind of expression, including arithmetic operators:

> 一门编程语言最有用的功能莫过于能够用一个个小模块来拼接创作。例如函数的参数可以是任何一种表达式，包括代数运算符：

```
x = math.sin(degrees / 360.0 * 2 * math.pi) ß
```

And even function calls:

> 再或者函数的调用本身也可以作为参数：

```
x = math.exp(math.log(x+1))
```

Almost anywhere you can put a value, you can put an arbitrary expression, with one exception: the left side of an assignment statement has to be a variable name. Any other expression on the left side is a syntax error (we will see exceptions to this rule later).

> 你可以在任何地方放一个值，放任何一个表达式，只有一个例外：一个声明语句的左边必须是变量名。任何其他的表达式放到等号左边都会导致语法错误（当然也有例外，等会再给介绍）。

```
>>> minutes = hours * 60                # right
>>> minutes = hours * 60                # right
>>> hours * 60 = minutes                # wrong!
>>> hours * 60 = minutes                # wrong!
SyntaxError: can't assign to operator
```

> 译者注：上述例子里面把表达式复制为变量是不行的，所说的例外估计是指尤达大师命名法，这个后面看到再说。

3.4 Adding new functions 添加新函数

So far, we have only been using the functions that come with Python, but it is also possible to add new functions. A function definition specifies the name of a new function and the sequence of statements that run when the function is called. Here is an example:

> 目前我们学到了一些Python自带的函数，自己添加新的函数也是可以的。函数定义要指定这个心函数的名字，还需要一系列语句放到这个函数里面，当调用这个函数的时候，就会运行这些语句了。

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print("I sleep all night and I work all day.")
```

def is a keyword that indicates that this is a function definition. The name of the function is print_lyrics. The rules for function names are the same as for variable names: letters, numbers and underscore are legal, but the first character can't be a number. You can't use a keyword as the name of a function, and you should avoid having a variable and a function with the same name.

> 这里的def就是一个关键词，意思是这是在定义一个函数。函数的名字就是print_lyrics，函数的命名规则和变量命名规则基本差不多，都是字幕梳子或者下划线，但是不能用数字打头。另外也不能用关键词做函数名，还要注意尽量避免函数名和变量名发生重复。

The empty parentheses after the name indicate that this function doesn't take any arguments.

> 函数名后面的括号是空的，意思是这个函数不需要参数。

The first line of the function definition is called the header; the rest is called the body. The header has to end with a colon and the body has to be indented. By convention, indentation is always four spaces. The body can contain any number of statements.

> 函数定义的第一行叫做头部，剩下的叫做函数体。函数头部的末尾必须有一个冒号，函数体必须是相对函数头部有缩进的，距离行首相对于函数头要有四个空格的距离。函数体可以有任意长度的语句。

（译者注：缩进是Python最强制的要求，本书的翻译用的MarkDown在生成的时候可能未必能够完美缩进，所以大家多注意一下自己调整哈，这个超级重要！） The strings in the print statements are enclosed in double quotes. Single quotes and double quotes do the same thing; most people use single quotes except in cases like this where a single quote (which is also an apostrophe) appears in the string.

> 在打印语句中，要打印的字符串需要用双引号括着。单引号和双引号效果一样，除非是字符串中已经出现了单引号，大家一般都是用单引号的。

All quotation marks (single and double) must be “straight quotes”, usually located next to Enter on the keyboard. “Curly quotes”, like the ones in this sentence, are not legal in Python.

> 所有的引号都必须是键盘上直接是引号的那个"键，无论是单引号还是双引号。就是回车键左边那个。“Curly quotes”这种引号，在Python里面是非法的。

If you type a function definition in interactive mode, the interpreter prints dots (...) to let you know that the definition isn’t complete:

> 如果你在交互模式下面定义函数，解释器会显示三个小点来提醒你定义还没有完成：

```
>>> def print_lyrics():
>>> def print_lyrics():
...
print("I'm a lumberjack, and I'm okay.") ...
print("I sleep all night and I work all day.") ...
```

To end the function, you have to enter an empty line. Defining a function creates a function object, which has type function:

> 在函数定义完毕的结尾，必须输入一行空白行。定义函数会创建一个函数类的对象，有type函数。

```
>>> print(print_lyrics)
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> type(print_lyrics)
>>> type(print_lyrics)
<class 'function'>
```

The syntax for calling the new function is the same as for built-in functions:

> 调用新函数的语法和调用内置函数是一样的：

```
>>> print_lyrics()
>>> print_lyrics()
I'm a lumberjack, and I'm okay. I sleep all night and I work all day.
```

Once you have defined a function, you can use it inside another function. For example, to repeat the previous refrain, we could write a function called repeat_lyrics:

> 一旦你定义了一个函数，就可以在其它函数里面来调用这个函数。比如咱们重复一下刚刚讨论的，写一个叫做重repeat_lyrics的函数。

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

And then call repeat_lyrics:

> 然后调用一下这个函数：

```
>>> repeat_lyrics()
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay. I sleep all night and I work all day. I'm a lumberjack, a
```

But that's not really how the song goes.

> 当然了，实际这首歌可不是这样的哈。

3.5 Definitions and uses 定义并使用

Pulling together the code fragments from the previous section, the whole program looks like this:

> 把前面这些小块的代码来整合一下，整体上程序看着大概是这样的：

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print("I sleep all night and I work all day.")
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
repeat_lyrics()
```

This program contains two function definitions: print_lyrics and repeat_lyrics. Function definitions get executed just like other statements, but the effect is to create function objects. The statements inside the function do not run until the function is called, and the function definition generates no output.

> 这个程序包含两个函数的定义：print_lyrics以及repeat_lyrics，函数定义的执行就和其他语句一样，但是效果是创建函数对象。函数定义中的语句直到函数被调用的时候才会运行，函数的定义本身不会有任何输出。

As you might expect, you have to create a function before you can run it. In other words, the function definition has to run before the function gets called.

> 如你所愿了，你可以建立一个函数，然后运行一下试试了。换种说法就是，在调用之前一定要先把函数定义好。

As an exercise, move the last line of this program to the top, so the function call appears before the definitions. Run the program and see what error message you get. Now move the function call back to the bottom and move the definition of print_lyrics after the definition of repeat_lyrics. What happens when you run this program?

> 作为练习，把这个程序的最后一行放到顶部，这样函数调用就在函数定义之前了。运行一下看看出错的信息是什么。
>
> 然后再把函数调用放到底部，把print_lyrics这个函数的定义放到repeat_lyrics这个函数的后面。再看看这次运行会出现什么样子？

3.6 Flow of execution 运行流程

To ensure that a function is defined before its first use, you have to know the order statements run in, which is called the flow of execution.

> 为了确保一个函数在首次被调用之前已经定义，你必须要之道语句运行的顺序，也就是所谓『运行流程』。

Execution always begins at the first statement of the program. Statements are run one at a time, in order from top to bottom.

> 一个Python程序都是从第一个语句开始运行的。从首至尾，每次运行一个语句。

Function definitions do not alter the flow of execution of the program, but remember that statements inside the function don't run until the function is called.

> 函数的定义并不会改变程序的运行流程，但要注意，函数体内部的语句只有在函数被调用的时候才会运行。

A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, runs the statements there, and then comes back to pick up where it left off.

> 函数调用就像是运行流程有了绕道的行为。没有直接去执行下一个语句，运行流跳入到函数体内，运行里面的语句，然后再回来从离开的地方继续执行。

That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to run the statements in another function. Then, while running that new function, the program might have to run yet another function!

> 这么解释很简明易懂了，只要你记住一个函数可以调用另一个就行。在一个函数的中间，程序有可能必须运行一下其他函数中的语句。所以运行新的函数的时候，程序可能也必须运行其他的函数！

译者注：看着很绕嘴，其实蛮简单的，就是跳出跳入互相调用而已。 Fortunately, Python is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

> 幸运的是，Python很善于追踪应该执行的位置，所以每次一个函数执行完毕了，程序都会回到当时跳出的位置，然后继续运行。等执行到了程序的末尾，就终止了。

In summary, when you read a program, you don't always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.

> 总的来说，你阅读一个程序的时候，并不一定总是要从头到尾来读的。有时候你要按照运行流程来读才更好理解。

3.7 Parameters and arguments 形式参数和实际参数

译者注：这里提到的形参和实参实际上是传值方式的区别，这个在最基本的编程入门课程中老师应该都比较强调的。实际参数就是调用函数时候传给他的那个参数；而形式参数可以理解为函数内部定义用的参数。老外对这个的思辩也很多。这里我先不说太多，翻译着再看。大家可以去网上多搜索一下，比如在StackOverflow和MSDN Some of the functions we have seen require arguments. For example, when you call math.sin you pass a number as an argument. Some functions take more than one argument: math.pow takes two, the base and the exponent.

> 我们已经看到了一些函数了，他们都需要实际参数。比如当你调用数学的正弦函数的时候你就需要给它一个数值作为实际参数。有的函数需要一个以上的实际参数，比如幂指数函数需要两个，一个是底数，一个是幂次。

Inside the function, the arguments are assigned to variables called parameters. Here is a definition for a function that takes an argument:

> 在函数里面，实际参数会被赋值给形式参数。下面就是一个使用单个实际参数的函数的定义：

```
def print_twice(bruce):
    print(bruce)
    print(bruce)
```

This function assigns the argument to a parameter named bruce. When the function is called, it prints the value of the parameter (whatever it is) twice. This function works with any value that can be printed.

> 这个函数把传来的实际参数的值赋给了一个名字叫做burce的形式参数。当函数被调用的时候，就会打印出形式参数的值两次（无论是什么内容）。任何能打印的值都适用于这个函数。

```
>>> print_twice('Spam')
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(42)
>>> print_twice(42)
42
42
>>> print_twice(math.pi)
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

The same rules of composition that apply to built-in functions also apply to programmer-defined functions, so we can use any kind of expression as an argument for print_twice: 适用于Python内置函数的组合规则对自定义的函数也是适用的，所以我们可以把表达式作为实际参数：

```
>>> print_twice('Spam '*4)
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

The argument is evaluated before the function is called, so in the examples the expressions 'Spam '*4 and math.cos(math.pi) are only evaluated once. You can also use a variable as an argument:

> 实际参数在函数被调用之前要先被运算一下，所以上面例子中作为实际参数的两个表达式都是在print_twice函数调用之前仅计算了一次。
>
> 当然了，也可以用变量做实际参数了：

```
>>> michael = 'Eric, the half a bee.'
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
>>> print_twice(michael)
Eric, the half a bee. Eric, the half a bee.
```

The name of the variable we pass as an argument (michael) has nothing to do with the name of the parameter (bruce). It doesn't matter what the value was called back home (in the caller); here in print_twice, we call everybody bruce.

> 咱们传递给函数的这个实际参数是一个变量，这个变量名michael和函数内部的形式参数bruce没有任何关系。在程序主体内部参数传过去就行了，参数名字对于函数内部没有作用；比如在这个print_twice函数里面，任何传来的值，在这个print_twice函数体内，都被叫做bruce。

译者注：这里要跟大家解释一下，传递参数的时候用的是实际参数，是把这个实际参数的值交给调用的函数，函数内部接收这个值，可以命名成任意其他名字的形式参数，差不多就这么个意思了。

3.8 Variables and parameters are local 函数内部变量和形参都是局部的

When you create a variable inside a function, it is local, which means that it only exists inside the function. For example:

> 在函数内部建立一个变量，这个变量是仅在函数体内部才存在。例如：

```
def cat_twice(part1, part2):
    cat = part1 + part2
    print_twice(cat)
```

This function takes two arguments, concatenates them, and prints the result twice. Here is an example that uses it:

> 这个函数得到两个实参，把它们连接起来，然后调用print_twice函数来输出结果两次。

```
>>> line1 = 'Bing tiddle '
>>> line1 = 'Bing tiddle '
>>> line2 = 'tiddle bang.'
>>> line2 = 'tiddle bang.'
>>> cat_twice(line1, line2)
>>> cat_twice(line1, line2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
```

When cat_twice terminates, the variable cat is destroyed. If we try to print it, we get an exception:

> 当cat_twice运行完毕了，这个名字叫做cat的变量就销毁了。咱们再尝试着打印它一下，就会得到异常：

```
>>> print(cat)
>>> print(cat)
NameError: name 'cat' is not defined
```

Parameters are also local. For example, outside print_twice, there is no such thing as bruce.

> 形式参数也是局部起作用的。例如在print_twice这个函数之外，是不存在bruce这个变量的。

译者注：当然你可以在函数外定义一个同名变量叫做bruce，但这两个没有关系，大家可以动手自己试试，这也是作者所鼓励的一种探索思维。

3.9 Stack diagrams 栈图

To keep track of which variables can be used where, it is sometimes useful to draw a stack diagram. Like state diagrams, stack diagrams show the value of each variable, but they also show the function each variable belongs to.

> 要追踪一个变量能在哪些位置使用，咱们就可以画个图表来实现，这种图表叫做栈图。栈图和我们之前提到的状态图有些相似，也会表征每个变量的值，不同的是栈图还会标识出每个变量所属的函数。

Each function is represented by a frame. A frame is a box with the name of a function beside it and the parameters and variables of the function inside it. The stack diagram for the previous example is shown in Figure 3.1.

> 每个函数都用一个框架来表示。框架的边上要标明函数的名字，框内填写函数内部的形参和变量。上文中样例代码的栈图如下图3.1所示。

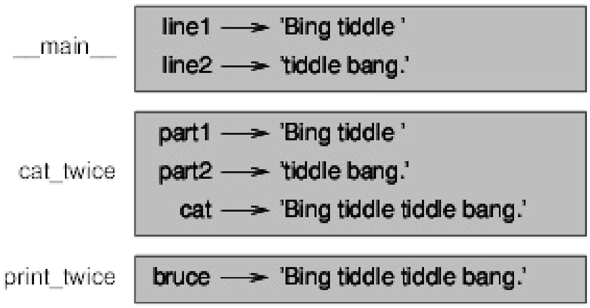


Figure 3.1: Stack diagram.

> 图3.1 栈图

The frames are arranged in a stack that indicates which function called which, and so on. In this example, print*twice was called by cat_twice, and cat_twice was called by _main__*, which is a special name for the topmost frame. When you create a variable outside of any function, it belongs to__main__.

> 一个栈中的这些框也表示了函数调用的关系等等。在上面这个例子中，print*twice*被*cat_twice*调用了两次，而*cat_twice*被_main__这个函数调用。__main__这个函数很特殊，属于最外层框架，也被叫做主函数。当你在所有函数之外建立一个变量的时候，这个变量就属于主函数所有。

Each parameter refers to the same value as its corresponding argument. So,part1 has the same value as line1, part2 has the same value as line2, and bruce has the same value as cat.

> 每个形式参数都保存了所对应的实际参数的值。因此part1的值和line1一样，part2的值就和line2一样，同理可知bruce的值就和cat一样了。

If an error occurs during a function call, Python prints the name of the function, the name of the function that called it, and the name of the function that called that, all the way back to __main__.

> 如果函数调用的时候出错了，Python会打印出这个出错函数的名字，调用这个出错函数的函数名，以及调用这个调用了出错函数的函数的函数名，一直追溯到主函数。（译者注：好绕口哈。。。就是会溯源回去啦。）

For example, if you try to access cat from within print_twice, you get a NameError:

> 例如，如果你想在print_twice这个函数中读取cat的值，就会得到一个变量名错误：

```
Traceback (innermost last):
File "test.py", line 13, in __main__
cat_twice(line1, line2)
File "test.py", line 5, in cat_twice
print_twice(cat)
File "test.py", line 9, in print_twice
print(cat)
NameError: name 'cat' is not defined
```

This list of functions is called a traceback. It tells you what program file the error occurred in, and what line, and what functions were executing at the time. It also shows the line of code that caused the error.

> 这个一系列的函数列表，就是一个追溯了。这回告诉你哪个程序文件出了错误，哪一行出了错误，以及当时哪些函数在运行。还会告诉你引起错误的代码所在行号。（译者注：这个简直太棒了，大家一定要留心这个功能以及出错提示，以后要用来解决很多bug呢。）

The order of the functions in the traceback is the same as the order of the frames in the stack diagram. The function that is currently running is at the bottom.

> 追溯中对函数顺序的排列是同栈图的方框顺序一样的。当前运行的函数会放在最底部。

3.10 Fruitful functions and void functions 有返回值的函数 和 无返回值的函数

Some of the functions we have used, such as the math functions, return results; for lack of a better name, I call them fruitful functions. Other functions, like print_twice, perform an action but don't return a value. They are called void functions.

> 咱们用过的一些函数，比如数学的函数，都会返回各种结果；也没别的好名字，就叫他们有返回值函数。其他的函数，比如print_twice，都是进行一些操作，但不返回值。那就叫做无返回值函数好了。

When you call a fruitful function, you almost always want to do something with the result; for example, you might assign it to a variable or use it as part of an expression:

> 当你调用一个有返回值的函数的时候，一般总是要利用一下结果的；比如，你可能需要把结果赋值给某个变量，然后在表达式里面来使用一下：

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

When you call a function in interactive mode, Python displays the result:

> 当你在交互模式调用一个函数的时候，Python会显示结果：

```
>>> math.sqrt(5) 2.2360679774997898
>>> math.sqrt(5) 2.2360679774997898
```

But in a script, if you call a fruitful function all by itself, the return value is lost forever!

> 如果是脚本模式，你运行一个有返回值的函数，但没有利用这个返回值，这个返回值就会永远丢失了！（译者注：只要有返回值就一定要利用！）

```
math.sqrt(5)
```

This script computes the square root of 5, but since it doesn't store or display the result, it is not very useful.

> 这个脚本计算了5的平方根，但没存储下来，也没有显示出来，所以就根本没用了。

Void functions might display something on the screen or have some other effect, but they don't have a return value. If you assign the result to a variable, you get a special value called None.

> 无返回值的函数要么就是屏幕上显示出一些内容，要么就有其他的功能，但就是没有返回值。如果你把这种函数的结果返回给一个变量，就会的到特殊的值：空。

```
>>> result = print_twice('Bing')
>>> result = print_twice('Bing')
Bing Bing
>>> print(result)
>>> print(result)
None
```

The value None is not the same as the string 'None'. It is a special value that has its own type:

> 这种None是空值的意思，和字符串'None'是不一样的。是一种特殊的值，并且有自己的类型。（译者注，就相当于null了。）

```
>>> print(type(None))
>>> print(type(None))
<class 'NoneType'>
```

The functions we have written so far are all void. We will start writing fruitful functions in a few chapters.

> 我们目前为止写的函数还都是无返回值的。接下来的新的章节里面，咱们就要开始写一些有返回值的函数了。

3.11 Why functions? 为啥要用函数？

It may not be clear why it is worth the trouble to divide a program into functions. There are several reasons:

> 为什么要费这么多力气来把程序划分成一个个函数呢？这么麻烦值得么？原因如下：

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read and debug.

> - 创建一个新的函数，你就可以把一组语句用一个名字来命名，这样你的程序读起来就清晰多了，后期维护调试也方便。

- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.

> - 函数的出现能够避免代码冗余，程序内的一些重复的内容就会简化了，变得更小巧。而且在后期进行修改的时候，你只要改函数中的一处地方就可以了，很方便。

- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.

> - 把长的程序切分成一个个函数，你就可以一步步来debug调试，每次只应对一小部分就可以，然后把它们组合起来就可以用了。

- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

> - 精细设计的函数会对很多程序都有用处。一旦你写好了并且除了错，这种函数代码可以再利用。

3.12 Debugging 调试

One of the most important skills you will acquire is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

> 给程序调试是你应当掌握的最关键技能之一了。尽管调试的过程会有挫败感，也依然是最满足智力，最有挑战性，也是编程过程中最有趣的一个项目了。

In some ways debugging is like detective work. You are confronted with clues and you have to infer the processes and events that led to the results you see.

> 某种程度上，调试像是侦探工作一样。你面对着很多线索，必须推断出导致当前结果的整个过程和事件。

Debugging is also like an experimental science. Once you have an idea about what is going wrong, you modify your program and try again. If your hypothesis was correct, you can predict the result of the modification, and you take a step closer to a working program.

> 调试也有点像一门实验科学。一旦你有了一个关于所出现的错误的想法，你就修改一下程序再试试看。如果你的假设是正确的，你就能够预料到修改导致的结果，这样在编程的水平上，你就上了一层台阶了，距离让程序工作起来也更近了。

If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out, “When you have eliminated the impossible, whatever remains, however improbable, must be the truth.” (A. Conan Doyle, The Sign of Four)

> 如果你的推测是错误的，你必须提出新的来。就像夏洛克.福尔摩斯之处的那样，『当你剔除了所有那些不可能，剩下的无论多么荒谬，都必然是真相。』（引自柯南道尔的小说《福尔摩斯探案：四签名》）

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should start with a working program and make small modifications, debugging them as you go.

> 对于一些人来说，编程和调试是一回事。也就是说，编程就是对一个程序逐渐进行调试，一直到程序按照设想工作为止。这种思想意味着你要从一段能工作的程序来起步，一点点做小修改和调试。

For example, Linux is an operating system that contains millions of lines of code, but it started out as a simple program Linus Torvalds used to explore the Intel 80386 chip. According to Larry Greenfield, “One of Linus’s earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved to Linux.” (The Linux Users’ Guide Beta Version 1).

> 例如，Linux是一个有上百万行代码的操作系统，但最早它起源于Linus Torvalsd的一段小代码。这个小程序是作者用来探索Intel的80386芯片的。根据Larry Greenfield回忆，『Linus早起的项目就是很小的一个程序，这个程序能够在输出AAAA和BBBB之间进行转换。这后来就发展除了Linux了。』（引用自Linux用户参考手册beta1版）

3.13 Glossary 术语列表

function: A named sequence of statements that performs some useful operation. Functions may or may not take arguments and may or may not produce a result.

> 函数：一系列有序语句的组合，有自己的名字，并且用在某些特定用途。可以要求输入参数，也可以没有参数，可以返回值，也可以没有返回值。

function definition: A statement that creates a new function, specifying its name, parameters, and the statements it contains.

> 函数定义：创建新函数的语句，确定函数的名字，形式参数，以及函数内部的语句。

function object: A value created by a function definition. The name of the function is a variable that refers to a function object.

> 函数对象：由函数定义所创建的值，函数名字指代了这一函数对象。

header: The first line of a function definition.

> 函数头：函数定义的第一行。

body: The sequence of statements inside a function definition.

> 函数体：函数定义内部的一系列有序语句。

parameter: A name used inside a function to refer to the value passed as an argument.

> 形式参数：用来在函数内部接收实际参数传来的值，并被函数在函数内部使用。

function call: A statement that runs a function. It consists of the function name followed by an argument list in parentheses.

> 函数调用：运行某个函数的语句。包括了函数的名字以及括号，括号内放函数需要的实际参数。

argument: A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function.

> 实际参数：当函数被调用的时候，提供给函数的值。这个值会被函数接收，赋给函数内部的形式参数。

local variable: A variable defined inside a function. A local variable can only be used inside its function.

> 局部变量：函数体内定义的变量。局部变量只在函数内部有效。

return value: The result of a function. If a function call is used as an expression, the return value is the value of the expression.

> 返回值：函数返回的结果。如果一个函数调用被用作了表达式，这个返回值就是这个表达式所代表的值。

fruitful function: A function that returns a value.

> 有返回值函数：返回一个值作为返回值的函数。

void function: A function that always returns None.

> 无返回值函数：不返回值，只返回一个空None的函数。

None: A special value returned by void functions.

> 空值：无返回值函数所返回的一种特殊的值。

module: A file that contains a collection of related functions and other definitions.

> 模块：包含一系列相关函数以及其他一些定义的文件。

import statement: A statement that reads a module file and creates a module object.

> 导入语句：读取模块并且创建一个模块对象的语句。

module object: A value created by an import statement that provides access to the values defined in a module.

> 模块对象：导入语句创建的一个值，允许访问模块所定义的值。

dot notation: The syntax for calling a function in another module by specifying the module name followed by a dot (period) and the function name.

> 点符号：调用某一个模块的某一函数的语法形式，就是模块名后加一个点，也就是英文的句号，再加函数名。

composition: Using an expression as part of a larger expression, or a statement as part of a larger statement.

> 组合：把表达式作为更大的表达式的一部分，或者把语句作为更大语句的一部分。

flow of execution: The order statements run in.

> 运行流程：语句运行的先后次序。

stack diagram: A graphical representation of a stack of functions, their variables, and the values they refer to.

> 栈图：对函数关系、变量内容及结构的图形化表示。

frame: A box in a stack diagram that represents a function call. It contains the local variables and parameters of the function.

> 框架：栈图中的方框，表示了一次函数调用。包括函数的局部变量和形式参数。

traceback: A list of the functions that are executing, printed when an exception occurs.

> 追踪：对运行中函数的列表，当有异常的时候就会输出。

3.14 Exercises 练习

Exercise 1 练习1

Write a function named right_justify that takes a string named s as a parameter and prints the string with enough leading spaces so that the last letter of the string is in column 70 of the display.

> 写一个名叫right_justify的函数，形式参数是名为s的字符串，将字符串打印，前面流出足够的空格，让字符串最后一个字幕在第70列显示。

```
>>> right_justify('monty')
>>> right_justify('monty')
```

Hint: Use string concatenation and repetition. Also, Python provides a built-in function called len that returns the length of a string, so the value of len('monty') is 5.

> 提示：使用字符拼接和重复来实现。另外Python还提供了内置的名字叫做len的函数，可以返回一个字符串的长度，比如len('monty')的值就是5了。

Exercise 2 练习2

A function object is a value you can assign to a variable or pass as an argument. For example, do_twice is a function that takes a function object as an argument and calls it twice:

> 你可以把一个函数对象作为一个值赋给一个变量或者作为一个实际参数来传递给其他函数。比如，do_twice就是一个把其他函数对象当做参数的函数，它的功能是调用对象函数两次：

```
def do_twice(f):
f()
f()
```

Here's an example that uses do_twice to call a function named print_spam twice.

> 下面是另一个例子，这里用了do_twice来调用一个名叫print_spam的函数两次。

```
def print_spam():
print('spam')
do_twice(print_spam)
```

1.Type this example into a script and test it.

> 把上面的例子写成脚本然后试一下。

2.Modify do_twice so that it takes two arguments, a function object and a value, and calls the function twice, passing the value as an argument.

> 修改一下do_twice这个函数，让它接收两个实际参数，一个是函数对象，一个是值，调用对象函数两次，并且赋这个值给对象函数作为实际参数。

3.Copy the definition of print_twice from earlier in this chapter to your script.

> 把print_twice这个函数的定义复制到你的脚本里面，去本章开头找一下这个例子哈。

4.Use the modified version of do_twice to call print_twice twice, passing'spam' as an argument.

> 用修改过的这个do_twice来调用print_twice两次，用字符串『spam』传递过去作为实际参数。

5.Define a new function called do_four that takes a function object and a value and calls the function four times, passing the value as a parameter. There should be only two statements in the body of this function, not four.

> 定义一个新的函数，名字叫做do_four，使用一个函数对象和一个值作为实际参数，调用这个对象函数四次，传递这个值作过去为对象函数的一个形式参数。这个函数体内只要有两个语句就够了，而不是四个。

Solution

> 样例代码：

Exercise 3 练习三

Note: This exercise should be done using only the statements and other features we have learned so far.

> 注意：这个练习应该只用咱们目前学习过的语句和其他功能来实现。

1.Write a function that draws a grid like the following:

> 写一个函数，输出如下：

```
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
```

Hint: to print more than one value on a line, you can print a comma-separated sequence of values:

> 提示：要一次打印超过一行，可以用逗号分隔一下就能换行了。如下所示：

```
print('+', '-')
```

By default, print advances to the next line, but you can override that behavior and put a space at the end, like this:

> 默认情况下，print会打印到下一行，你可以手动覆盖掉这个行为，在末尾输出一个空格就可以了：

```
print('+', end=' ')
print('-')
```

The output of these statements is '+ -'.

> 上面的语句输出结果就是：'+ -'。

A print statement with no argument ends the current line and goes to the next line.

> 没有参数的print语句会把当前的行结束，去下一行。

2.Write a function that draws a similar grid with four rows and four columns.

> 写一个四行四列的小网格绘制的程序。

Solution 样例 Credit: This exercise is based on an exercise in Oualline, Practical C Programming, Third Edition, O’Reilly Media, 1997.

> 此练习基于Oualline的书《实践C语言编程》第三版，O'Reilly出版社，1997年版

Chapter 4 Case study: interface design 案例学习：交互设计

This chapter presents a case study that demonstrates a process for designing functions that work together.

> 本章会提供一个案例，用于展示如何却设计一些共同工作的函数。

It introduces the turtle module, which allows you to create images using turtle graphics. The turtle module is included in most Python installations, but if you are running Python using PythonAnywhere, you won't be able to run the turtle examples (at least you couldn't when I wrote this).

> 本章介绍了小乌龟这个模块，这允许你用小龟的图形功能来制作一些图形。乌龟模块在大部分的Python中都有安装，不过如果你在线使用PythnAnywhere，你就无法运行这些乌龟样例了（至少我写这本教材的时候还不行）。
>
> （译者注：都学到第四章了，你还不本地安装个Python也太说不过去了吧。）

If you have already installed Python on your computer, you should be able to run the examples. Otherwise, now is a good time to install. I have posted instructions at http://tinyurl.com/thinkpython2e. Code examples from this chapter are available from http://thinkpython2.com/code/polygon.py.

> 如果你已经安装了Python在你的电脑上，你就能运行这些例子了。没安装的话呢，这就是安装的好时机了呗。我已经把相关介绍放到网页上面了，点击访问。
>
> 本章代码样例可以点击此链接来下载了。

4.1 The turtle module 乌龟模块

To check whether you have the turtle module, open the Python interpreter and type:

> 要检查你是不是已经安装了这个乌龟模块，你要打开Python解释器来输入如下内容：

```
>>> import turtle
>>> import turtle
>>> bob = turtle.Turtle()
>>> bob = turtle.Turtle()
```

When you run this code, it should create a new window with small arrow that represents the turtle. Close the window.

> 运行上述例子的时候，应该就能新建一个小窗口，还有个小箭头象征小乌龟。如果有的话就对了，把窗口关掉吧先。

Create a file named mypolygon.py and type in the following code:

> 建立一个叫做mypolygon.py的文件，在里面输入如下内容：

```
import turtle
bob = turtle.Turtle()
print(bob)
turtle.mainloop()
```

The turtle module (with a lowercase 't') provides a function called Turtle(with an uppercase 'T') that creates a Turtle object, which we assign to a variable named bob. Printing bob displays something like:

> 这个小乌龟模块（记着是小写的t）提供了一个叫做Turtle（注意这里是大写的，大小写要去分！）的函数，这个函数会创建一个Turtle对象，我们把它赋值给bob这个变量。打印一下bob就能显示如下内容：

```
<turtle.Turtle object at 0xb7bfbf4c>
```

This means that bob refers to an object with type Turtle as defined in module turtle.

> 这就意味着bob已经指向了模块turtle中所定义的Turtle类的一个对象。

mainloop tells the window to wait for the user to do something, although in this case there's not much for the user to do except close the window.

> mainloop这个函数是告诉窗口等用户来做些事情，当然本次尝试的情况下用户也就是关闭窗口而已了。

Once you create a Turtle, you can call a method to move it around the window. A method is similar to a function, but it uses slightly different syntax. For example, to move the turtle forward:

> 一旦你创建了一个Trutle，你就可以调用一些方法让他在窗口中移动。方法跟函数有点相似，但语法的使用稍微不太一样。比如你可以让小乌龟往前走：

```
bob.fd(100)
```

The method, fd, is associated with the turtle object we're calling bob. Calling a method is like making a request: you are asking bob to move forward.

> fd这个方法，是turtle类这个叫做bob的对象所包含的。调用这个方法就像是做出一个请求一样：你再让bob向前移动。

The argument of fd is a distance in pixels, so the actual size depends on your display.

> fd这个方法的参数是像素数距离，所以实际的大小依赖于你显示器的情况了。

Other methods you can call on a Turtle are bk to move backward, lt for left turn, and rt right turn. The argument for lt and rt is an angle in degrees.

> Turtle对象中还有一些其他方法，比如bk是后退，lt是左转，rt是右转。lt和rt用偏转角度做参数。

Also, each Turtle is holding a pen, which is either down or up; if the pen is down, the Turtle leaves a trail when it moves. The methods pu and pd stand for “pen up” and “pen down”.

> 另外，每个Turtle都相当于带着笔，可以落下或者抬起；如果笔落下了，Turtle移动的时候就会留下轨迹了。抬笔落笔的方法缩写粉笔嗯是pu和pd。

To draw a right angle, add these lines to the program (after creating bob and before calling mainloop):

> 画一个直角，就要把下面这些线加到程序里面（当然要先创建一个bob并且在此之前运行mainloop）：

```
bob.fd(100)
bob.lt(90)
bob.fd(100)
```

When you run this program, you should see bob move east and then north, leaving two line segments behind.

> 运行这个程序，你就能看到bob先向东再往北，后面就留下了两根互相垂直的线段了。

Now modify the program to draw a square. Don’t go on until you’ve got it working!

> 现在修改一下程序，去画一个正方形。这个程序运行不好的话就不要继续后面的章节！

4.2 Simple repetition 简单的重复

Chances are you wrote something like this:

> 你估计会写出如下的内容：

```
bob.fd(100)
bob.lt(90)
bob.fd(100)
bob.lt(90)
bob.fd(100)
bob.lt(90)
bob.fd(100)
```

We can do the same thing more concisely with a for statement. Add this example to mypolygon.py and run it again:

> 上面这个太麻烦了，咱们可以用一个for语句来让这个过程更简洁。把下面的代码添加到mypolygon.py中然后运行一下：

```
for i in range(4):
    print('Hello!')
```

You should see something like this:

> 你将会看到这样的输出：

```
Hello!
Hello!
Hello!
Hello!
```

This is the simplest use of the for statement; we will see more later. But that should be enough to let you rewrite your square-drawing program. Don't go on until you do.

> 这就是for语句的最简单的一种应用；以后我们会看到更多。不过当前这种简单的足够你来重构一下你的正方形绘制程序了。不达目的不罢休，不要跳过困难哈，一定要编写出来这个再进行后面的内容。

Here is a for statement that draws a square:

> 这就是一个用for语句来画正方形的语句：

```
for i in range(4):
    bob.fd(100)
    bob.lt(90)
```

The syntax of a for statement is similar to a function definition. It has a header that ends with a colon and an indented body. The body can contain any number of statements.

> for语句的语法跟函数定义有点相似。有一个头部，头部的结尾要用冒号，然后还有一个缩进的循环体。循环体可以包含任意多的语句。

A for statement is also called a loop because the flow of execution runs through the body and then loops back to the top. In this case, it runs the body four times.

> for语句也被叫做循环，因为运行流程会重复执行循环体。在本节的例子中，循环进行了四次。

This version is actually a little different from the previous square-drawing code because it makes another turn after drawing the last side of the square. The extra turn takes more time, but it simplifies the code if we do the same thing every time through the loop. This version also has the effect of leaving the turtle back in the starting position, facing in the starting direction.

> 这次的正方形绘制代码实际上和之前的少有不同了，因为在画完了最后一个边之后，多了一次转向。多出来的这部分需要消耗额外的时间，但简化了下次我们来循环进行绘制的过程。这个版本的代码也有一个额外的效果：让小乌龟回到起点，朝着初始方向。

4.3 Exercises 练习

The following is a series of exercises using TurtleWorld. They are meant to be fun, but they have a point, too. While you are working on them, think about what the point is.

> 下面是一系列使用TurtleWorld的练习。主要就是比较有意思，不过也有一些训练的作用。你做这些练习的时候，一定要注意考虑这些训练的作用。

The following sections have solutions to the exercises, so don't look until you have finished (or at least tried).

> 练习后面是有一些样例的解决方案的，所以你要做完了再往后看，至少你得试试，不会做了看看答案也行哈。

1.Write a function called square that takes a parameter named t, which is a turtle. It should use the turtle to draw a square.

> 写一个函数叫做square（译者注：就是正方形的意思），有一个名叫t的参数，这个t是一个turtle。用这个turtle来画一个正方形。

Write a function call that passes bob as an argument to square, and then run the program again.

> 写一个函数调用，把bob作为参数传递给square，然后再运行这个程序。

2.Add another parameter, named length, to square. Modify the body so length of the sides is length, and then modify the function call to provide a second argument. Run the program again. Test your program with a range of values for length.

> 给这个square函数再加一个参数，叫做length（译者注：长度）。把函数体修改一下，让长度length赋值给各个边的长度，然后修改一下调用函数的代码，再提供一个这个对应长度的参数。再次运行一下，用一系列不同的长度值来测试一下你的程序。

3.Make a copy of square and change the name to polygon. Add another parameter named n and modify the body so it draws an n-sided regular polygon. Hint: The exterior angles of an n-sided regular polygon are 360/n degrees.

> 复制一下square这个函数，把名字改成polygon（译者注：意思为多边形）。另外添加一个参数叫做n，然后修改函数体，让函数实现画一个正n边的多边形。提示：正n多边形的外角为360/n度。

4.Write a function called circle that takes a turtle, t, and radius, r, as parameters and that draws an approximate circle by calling polygon with an appropriate length and number of sides. Test your function with a range of values of r.

> 在写一个叫做circle（译者注：圆）的函数，也用一个turtle类的对象t，以及一个半径r，作为参数，画一个近似的圆，通过调用polygon函数来近似实现，用适当的边长和边数。用不同的半径值来测试一下你的函数。

Hint: figure out the circumference of the circle and make sure that length * n = circumference.

> 提示：算出圆的周长，确保边长乘以边数的值（近似）等于圆周长。

5.Make a more general version of circle called arc that takes an additional parameter angle, which determines what fraction of a circle to draw. angle is in units of degrees, so when angle=360, arc should draw a complete circle.

> 在circle基础上做一个叫做arc的函数，在circle的基础上添加一个angle（译者注：角度）变量，用这个角度值来确定画多大的一个圆弧。用度做单位，当angle等于360度的时候，arc函数就应当画出一个整团了。

4.4 Encapsulation 封装

The first exercise asks you to put your square-drawing code into a function definition and then call the function, passing the turtle as a parameter. Here is a solution:

> 第一个练习让你把正方形绘制的代码定义到一个函数里面，然后调用这个函数，传入一个turtle对象作为参数。下面就是个例子了：

```
def square(t):
    for i in range(4):
        t.fd(100)
        t.lt(90)
square(bob)
```

The innermost statements, fd and lt are indented twice to show that they are inside the for loop, which is inside the function definition. The next line,square(bob), is flush with the left margin, which indicates the end of both the for loop and the function definition.

> 在最内部的语句里面，fd和lt缩进了两次，这个意思是他们是for循环的循环体内部成员，而for循环本身缩进了一次，说明for语句被包含在函数的定义当中。接下来的那行square(bob)，紧靠左侧，没有缩进，这说明for循环和函数定义都结束了。

Inside the function, t refers to the same turtle bob, so t.lt(90) has the same effect as bob.lt(90). In that case, why not call the parameter bob? The idea is that t can be any turtle, not just bob, so you could create a second turtle and pass it as an argument to square:

> 在函数体内部，t所指代的就是小乌龟bob，因此让t来左转九十度的效果完全等同于让bob来左转九十度。本文中没有把形式参数的名字设置成bob，这是为啥呢？是因为用t可以指代任意一个小乌龟，不仅仅是bob，所以你就能再创建另一个小乌龟，把它传递给square这个函数作为实际参数：

```
alice = Turtle()
square(alice)
```

Wrapping a piece of code up in a function is called encapsulation. One of the benefits of encapsulation is that it attaches a name to the code, which serves as a kind of documentation. Another advantage is that if you reuse the code, it is more concise to call a function twice than to copy and paste the body!

> 用函数的形式把一段代码包装起来，叫做封装。这样有一个好处，就是给代码起了个名字，有类似文档说明的功能，更好理解了。另外一个好处是下次重复使用这段代码的时候，再次调用函数就可以了，这比复制粘贴函数体可方便多了。

4.5 Generalization 泛化

The next step is to add a length parameter to square. Here is a solution:

> 下一步就是给square函数添加一个长度参数了。下面是样例：

```
def square(t, length):
    for i in range(4):
        t.fd(length)
        t.lt(90)
square(bob, 100)
```

Adding a parameter to a function is called generalization because it makes the function more general: in the previous version, the square is always the same size; in this version it can be any size.

> 给函数添加参数，就叫做泛化，因为者可以让函数的功能更广泛：在之前的版本中，square这个函数画出来的正方形总是一个尺寸的；在这个新版本里面，可以自定义边长了。

The next step is also a generalization. Instead of drawing squares, polygondraws regular polygons with any number of sides. Here is a solution:

> 下一步也还是泛化。这次就是不光要画正方形了，要画一个多边形，可以指定边数的。下面是样例：

```
def polygon(t, n, length):
    angle = 360 / n
    for i in range(n):
        t.fd(length)
        t.lt(angle)
polygon(bob, 7, 70)
```

This example draws a 7-sided polygon with side length 70.

> 这个例子画了一个每个边长度都为70像素的七边形。

If you are using Python 2, the value of angle might be off because of integer division. A simple solution is to compute angle = 360.0 / n. Because the numerator is a floating-point number, the result is floating point.

> 如果你用Python2的话，角度可能因为整除而导致的偏差。简单的解决方法就是用360.0来除以n而不是用360，这就是用浮点数替代了原来的整形，结果就是一个浮点数了。

When a function has more than a few numeric arguments, it is easy to forget what they are, or what order they should be in. In that case it is often a good idea to include the names of the parameters in the argument list:

> 当一个函数有超过一个数据参数的时候，很容易忘掉这些参数都是什么，或者忘掉他们的顺序。为了避免这个情况，可以把形式参数的名字包含在一个实际参数列表中：

```
polygon(bob, n=7, length=70)
```

These are called keyword arguments because they include the parameter names as “keywords” (not to be confused with Python keywords like whileand def).

> 这些列表叫做关键参数列表，因为他们把形式参数的名字作为关键词包含了进来。（注意区别这里的关键词可不是Python语言的关键词哈！这里就是字面意思，很关键的词。）

This syntax makes the program more readable. It is also a reminder about how arguments and parameters work: when you call a function, the arguments are assigned to the parameters.

> 这种语法结构让程序更容易被人读懂。也能提醒实际参数和形式参数的使用过程：调用一个函数的时候，把实际参数的值赋给了形式参数。

4.6 Interface design 界面设计

The next step is to write circle, which takes a radius, r, as a parameter. Here is a simple solution that uses polygon to draw a 50-sided polygon:

> 下一步就是写circle这个函数了，需要半径r作为一个参数。下面是一个简单的样例，使用polygon函数来画一个50边形，来接近一个圆：

```
import math
def circle(t, r):
    circumference = 2 * math.pi * r
    n = 50
    length = circumference / n
    polygon(t, n, length)
```

The first line computes the circumference of a circle with radius r using the formula 2 π r. Since we use math.pi, we have to import math. By convention,import statements are usually at the beginning of the script.

> 第一行计算了圆的周长，使用2乘以圆周率再乘以半径r。这个计算用到了圆周率，所以要导入math模块。通常都要把导入语句放到整个脚本的开头。

n is the number of line segments in our approximation of a circle, so length is the length of each segment. Thus, polygon draws a 50-sides polygon that approximates a circle with radius r.

> n是我们用来逼近一个圆所用的线段数量，所以length就是每一个线段的长度了。polygon画一个50边的多边形，来近似做一个半径为r的圆。

One limitation of this solution is that n is a constant, which means that for very big circles, the line segments are too long, and for small circles, we waste time drawing very small segments. One solution would be to generalize the function by taking n as a parameter. This would give the user (whoever calls circle) more control, but the interface would be less clean.

> 这种方案的一个局限性就是n是常数，就意味着对于一些大尺寸的圆，线段数目就太多了，而对小的圆，又浪费了很多小线段。解决的方法就是进一步扩展函数，让函数把n也作为一个参数。这就亏让用户（调用circle函数的任何人）有更多决定权，可以控制所用的线段数量，当然，界面就不那么简洁了。

The interface of a function is a summary of how it is used: what are the parameters? What does the function do? And what is the return value? An interface is “clean” if it allows the caller to do what they want without dealing with unnecessary details.

> 函数的界面就是关于它如何工作的一个概述：都有什么变量？函数实现什么功能？以及返回值是什么？允许调用者随意操作而不用处理一些无关紧要的细节，这种函数界面就是简洁的。

In this example, r belongs in the interface because it specifies the circle to be drawn. n is less appropriate because it pertains to the details of how the circle should be rendered.

> 在本节的例子中，r包含于界面内，因为要用它来确定所画圆的大小。n就不那么合适了，因为它是用来处理如何具体绘制一个圆的。

Rather than clutter up the interface, it is better to choose an appropriate value of n depending on circumference:

> 与其让界面复杂冗余，更好的思路是让n根据周长来自适应一个合适的值：

```
def circle(t, r):
    circumference = 2 * math.pi * r
    n = int(circumference / 3) + 1
    length = circumference / n
    polygon(t, n, length)
```

Now the number of segments is an integer near circumference/3, so the length of each segment is approximately 3, which is small enough that the circles look good, but big enough to be efficient, and acceptable for any size circle.

> 现在线段个数就是周长的三分之一了，因此每段线段的长度近似为3，这个大小可以让圆看着不错，也对任意大小的圆都适用了。

4.7 Refactoring 重构

When I wrote circle, I was able to reuse polygon because a many-sided polygon is a good approximation of a circle. But arc is not as cooperative; we can’t use polygon or circle to draw an arc.

> 当我写circle这个函数的时候，我能利用多边形函数polygon是因为一个足够多边的多边形和圆很接近。但圆弧就不太适合这个思路了；我们不能用多边形或者圆来画一个圆弧。

One alternative is to start with a copy of polygon and transform it into arc. The result might look like this:

> 一个替代的方法就是把polygon修改一下，转换成圆弧。结果大概如下所示：

```
def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = angle / n
    for i in range(n):
        t.fd(step_length)
        t.lt(step_angle)
```

The second half of this function looks like polygon, but we can’t reuse polygon without changing the interface. We could generalize polygon to take an angle as a third argument, but then polygon would no longer be an appropriate name! Instead, let’s call the more general function polyline:

> 这个函数的后半段看着和多边形那个还挺像的，但必须修改一下界面才能重利用多边形的代码。我们在多边形函数上增加angle（角度）作为第三个参数，但继续叫多边形就不太合适了，因为不闭合啊！所以就改名叫它多段线polyline：

```
def polyline(t, n, length, angle):
    for i in range(n):
    t.fd(length)
    t.lt(angle)
```

Now we can rewrite polygon and arc to use polyline:

> 现在就可以用多段线polyline来重写多边形polygon和圆弧arc：

```
def polygon(t, n, length):
    angle = 360.0 / n
    polyline(t, n, length, angle)
def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n
    polyline(t, n, step_length, step_angle)
```

Finally, we can rewrite circle to use arc:

> 最终，咱们就可以用圆弧arc来重写circle的实现了：

```
def circle(t, r):
    arc(t, r, 360)
```

This process—rearranging a program to improve interfaces and facilitate code re-use—is called refactoring. In this case, we noticed that there was similar code in arc and polygon, so we “factored it out” into polyline.

> 这个过程中，改进了界面设计，增强了代码再利用，这就叫做重构。在本节的这个例子中，我们先是注意到圆弧arc和多边形polygon有相似的代码，所以我们把他们都用多段线polyline来实现。

If we had planned ahead, we might have written polyline first and avoided refactoring, but often you don’t know enough at the beginning of a project to design all the interfaces. Once you start coding, you understand the problem better. Sometimes refactoring is a sign that you have learned something.

> 如果我们事先进行了计划，估计就会先写出多段线函数polyline，然后就不用重构了，但大家在开始一个项目之前往往不一定了解的那么清楚。一旦开始编码了，你就逐渐更理解其中的问题了。有时候重构就意味着你已经学到了新的内容了。

4.8 A development plan 开发计划

A development plan is a process for writing programs. The process we used in this case study is “encapsulation and generalization”. The steps of this process are:

> 开发计划是写程序的一系列过程。我们本章所用的就是『封装-泛化』的模式。这一过程的步骤如下：

1. Start by writing a small program with no function definitions.

> 开始写一个特别小的程序，没有函数定义。

1. Once you get the program working, identify a coherent piece of it, encapsulate the piece in a function and give it a name.

> 一旦有你的程序能用了，确定一下实现功能的这部分有练习的语句，封装成函数，并命名一下。

1. Generalize the function by adding appropriate parameters.

> 通过逐步给这个函数增加参数的方式来泛化。

1. Repeat steps 1–3 until you have a set of working functions. Copy and paste working code to avoid retyping (and re-debugging).

> 重复1-3步骤，一直到你有了一系列能工作的函数为止。把函数复制粘贴出来，避免重复输入或者修改了。

1. Look for opportunities to improve the program by refactoring. For example, if you have similar code in several places, consider factoring it into an appropriately general function.

> 看看是不是有通过重构来改进函数的可能。比如，假设你在一些地方看到了相似的代码，就可以把这部分代码做成一个函数。

This process has some drawbacks—we will see alternatives later—but it can be useful if you don't know ahead of time how to divide the program into functions. This approach lets you design as you go along.

> 这个模式有一些缺点，我们后续会看到一些替代的方式，但这个模式是很有用的，尤其对耐饿实现不值得怎么去把程序分成多个函数的情况。

4.9 docstring 文档字符串

A docstring is a string at the beginning of a function that explains the interface ("doc" is short for "documentation"). Here is an example:

> 文档字符串是指：在函数开头部位，解释函数的交互界面的字符串，doc是文档documentation的缩写。下面是一个例子：

```
def polyline(t, n, length, angle):
"""
Draws n line segments with the given length and     angle (in degrees) between them.
t is a turtle.     """
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

By convention, all docstrings are triple-quoted strings, also known as multiline strings because the triple quotes allow the string to span more than one line.

> 一般情况下，所有文档字符串都是三重引用字符串，也被叫做多行字符串，因为三重的单引号表示允许这个字符串是多行的。

It is terse, but it contains the essential information someone would need to use this function. It explains concisely what the function does (without getting into the details of how it does it). It explains what effect each parameter has on the behavior of the function and what type each parameter should be (if it is not obvious).

> 这些文字很简洁，但都包含了一些关键的信息，这些信息对于函数使用者来说至关重要。这些信息简要解释了函数的用途（不会说细节，也不会说如何实现）。文档解释了每个参数对函数行为的影响，以及各自的类型（一般在不是显而易见的情况下就给解释了）。

Writing this kind of documentation is an important part of interface design. A well-designed interface should be simple to explain; if you have a hard time explaining one of your functions, maybe the interface could be improved.

> 写这种文档，对交互界面的设计来说，是至关重要的。设计良好的交互界面应该很容易解释明白；如果你的函数有一个特别不好解释了，估计这个函数的交互设计还存在需要改进的地方。

4.10 Debugging 调试

An interface is like a contract between a function and a caller. The caller agrees to provide certain parameters and the function agrees to do certain work.

> 一个交互界面，就像是函数和调用者的一个中间人。调用者提供特定的参数，函数完成特定的任务。

For example, polyline requires four arguments: t has to be a Turtle; n has to be an integer; length should be a positive number; and angle has to be a number, which is understood to be in degrees.

> 例如，polyline这个多段线函数，需要四个实际参数：t必须是一个Turtle小乌龟；n（边数）必须是一个整形；length（长度）应该是一个正数；angle（角度）必须是一个以度为单位的角度值。

These requirements are called preconditions because they are supposed to be true before the function starts executing. Conversely, conditions at the end of the function are postconditions. Postconditions include the intended effect of the function (like drawing line segments) and any side effects (like moving the Turtle or making other changes).

> 这些要求叫做『前置条件』，因为要在函数开始运行之前就要实现才行。相应的在函数的结尾那里的条件叫『后置条件』。后置条件包含函数的预期效果（如画线段）和其他作用（如移动海龟或进行其他改动）。

Preconditions are the responsibility of the caller. If the caller violates a (properly documented!) precondition and the function doesn’t work correctly, the bug is in the caller, not the function.

> 前置条件是准备给函数调用者的。如果调用者违背了（妥当标注的）前置条件，然后函数不能正常工作，这个bug就会反馈在函数调用者上，而不是函数本身。

If the preconditions are satisfied and the postconditions are not, the bug is in the function. If your pre- and postconditions are clear, they can help with debugging.

> 如果前置条件得到了满足，而后置条件未能满足，这个bug就是函数的了。所以如果你的前后置条件都弄清晰，对调试很有帮助。

4.11 Glossary 术语列表

method: A function that is associated with an object and called using dot notation.

> 方法：某个类中一个对象所具有的函数，用点连接来进行调用。

loop: A part of a program that can run repeatedly.

> 循环：程序中重复运行的一部分。

encapsulation: The process of transforming a sequence of statements into a function definition.

> 封装：把一系列相关的语句整理定义成一个函数的过程。

generalization: The process of replacing something unnecessarily specific (like a number) with something appropriately general (like a variable or parameter).

> 泛化：把一些不必要的内容用更广泛通用的内容来替换掉的过程，比如把一个数字替换成了一个变量或者参数。

keyword argument: An argument that includes the name of the parameter as a "keyword".

> 关键词参数：一种特殊的实际参数，把形式参数的名字作为关键词包含在内。

interface: A description of how to use a function, including the name and descriptions of the arguments and return value.

> 交互界面：对如何使用一个函数的描述，包括了函数名，以及对实际参数和返回值的描述。

refactoring: The process of modifying a working program to improve function interfaces and other qualities of the code.

> 重构：对一份能工作的程序进行修改，改进函数交互界面以及提高代码其他方面质量的过程。

development plan: A process for writing programs.

> 开发计划：写程序的过程。

docstring: A string that appears at the top of a function definition to document the function's interface.

> 文档字符串：一个在函数定义的顶部的字符串，讲解函数的交互界面。

precondition: A requirement that should be satisfied by the caller before a function starts.

> 前置条件：函数开始之前，调用者应当满足的要求。

postcondition: A requirement that should be satisfied by the function before it ends.

> 后置条件：函数结束之前应该满足的一些要求。

4.12 Exercises 练习

Exercise 1 练习1

Download the code in this chapter from here.

> 点击下面这个链接下载代码。

1. Draw a stack diagram that shows the state of the program while executing circle(bob, radius). You can do the arithmetic by hand or add print statements to the code.

> 画一个栈图，表明运行函数circle(bob,radius)时候程序的状态。你可以手算一下，或者把输出语句加到代码上。

1. The version of arc in Section 4.7 is not very accurate because the linear approximation of the circle is always outside the true circle. As a result, the Turtle ends up a few pixels away from the correct destination. My solution shows a way to reduce the effect of this error. Read the code and see if it makes sense to you. If you draw a diagram, you might see how it works.

> 4.7小节中的那个版本的arc函数并不太精确，因为对圆进行线性逼近总会超过真实情况。结果就是小乌龟总会距离正确位置偏离一些像素。我的样例给出了一种降低这种误差程度的方法。阅读一下代码，看你能不能理解。如果你画一个图标，也许就能明白代码是怎么工作的了。

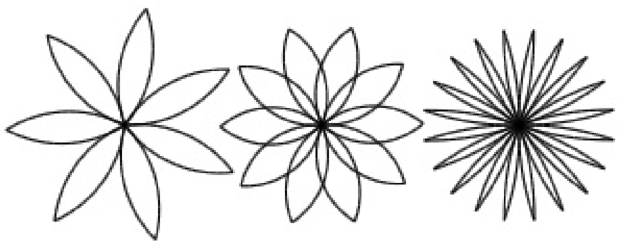


Figure 4.1: Turtle flowers.

Exercise 2 练习2

Write an appropriately general set of functions that can draw flowers as in Figure 4.1.

> 写一系列的合适的函数组合，画出图4.1所示的花图案。

Solution 样例： http://thinkpython2.com/code/flower.py, also requires 同时需要：http://thinkpython2.com/code/polygon.py.

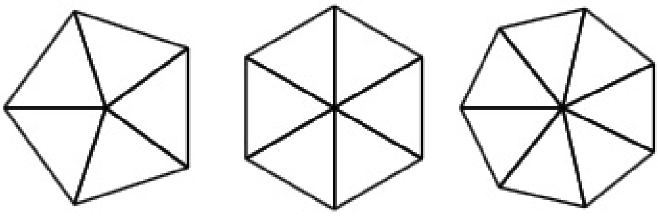


Figure 4.2: Turtle pies.

Exercise 3 练习3

Write an appropriately general set of functions that can draw shapes as in Figure 4.2.

> 写一系列的合适的函数组合，画出图4.2所示的形状。

Solution 样例： http://thinkpython2.com/code/pie.py.

Exercise 4 练习4

The letters of the alphabet can be constructed from a moderate number of basic elements, like vertical and horizontal lines and a few curves. Design an alphabet that can be drawn with a minimal number of basic elements and then write functions that draw the letters.

> 字母表当中的字母都可以用一定数量的基本元素来构建，比如竖直或者水平的线条，以及一些曲线。设计一个能用最小数量的基本元素画出来的字母表，然后写个函数来画字母出来。

You should write one function for each letter, with names draw_a, draw_b, etc., and put your functions in a file named letters.py. You can download a “turtle typewriter” from this link to help you test your code.

> 你应当为没一个字母写一个函数，名字就比如draw_a,draw_b等等，然后把你的函数放到一个叫做letters.py的文件中。你可以从这个链接下载一个乌龟打字机来帮你检测一下代码。

You can get a solution from here; it also requires this.

> 你可以参考这里的样例;同时还需要这些。

Exercise 5 练习5

Read about spirals at Wiki; then write a program that draws an Archimedian spiral (or one of the other kinds). Solution

> 去Wiki百科看一下螺旋线的相关内容;然后写个程序来画阿基米德曲线（曲线中的一种）。样例

Chapter 5 Conditionals and recursion 条件循环

The main topic of this chapter is the if statement, which executes different code depending on the state of the program. But first I want to introduce two new operators: floor division and modulus.

> 本章的主题是if语句，就是条件判断，会对应程序的不同状态来执行不同的代码。但首先我要介绍两种新的运算符：floor（地板除法，舍弃小数位）和modulus（求模，取余数）

5.1 Floor division and modulus 地板除和求模

The floor division operator, //, divides two numbers and rounds down to an integer. For example, suppose the run time of a movie is 105 minutes. You might want to know how long that is in hours. Conventional division returns a floating-point number:

> floor除法，中文有一种翻译是地板除法，挺难听，不过凑活了，运算符是两个右斜杠：//，与传统除法不同，地板除法会把运算结果的小数位舍弃，返回整值。例如，加入一部电影的时间长度是105分钟。你可能想要知道这部电影用小时来计算是多长。传统的除法运算如下，会返回一个浮点小数：

```
>>> minutes = 105
>>> minutes = 105
>>> minutes / 60
>>> minutes / 60
1.75
```

But we don't normally write hours with decimal points. Floor division returns the integer number of hours, dropping the fraction part:

> 不过一般咱们不写有小数的小时数。地板除法返回的就是整的小时数，舍弃掉小数位：

```
>>> minutes = 105
>>> minutes = 105
>>> hours = minutes // 60
>>> hours = minutes // 60
>>> hours
>>> hours
1
```

To get the remainder, you could subtract off one hour in minutes:

> 想要知道舍弃那部分的长度，可以用分钟数减去这么一个小时，然后剩下的分钟数就是了：

```
>>> remainder = minutes - hours * 60
>>> remainder = minutes - hours * 60
>>> remainder
>>> remainder
45
```

An alternative is to use the modulus operator, %, which divides two numbers and returns the remainder.

> 另外一个方法就是使用求模运算符了，百分号%就是了，求模运算就是求余数，会把两个数相除然后返回余数。

```
>>> remainder = minutes % 60
>>> remainder = minutes % 60
>>> remainder
>>> remainder
45
```

The modulus operator is more useful than it seems. For example, you can check whether one number is divisible by another—if x % y is zero, then x is divisible by y.

> 求模运算符的作用远不止如此。比如你可以用求模来判断一个数能否被另一个数整除——比如x%y如果等于0了，那就是意味着x能被y整除了。

Also, you can extract the right-most digit or digits from a number. For example, x % 10 yields the right-most digit of x (in base 10). Similarly x % 100yields the last two digits.

> 另外你也可以从一个数上取最右侧的一位或多位数字。比如，x%10就会得出x最右边的数字，也就是x的个位数字。同样的道理，用x%100得到的就是右面两位数字了。

If you are using Python 2, division works differently. The division operator, /, performs floor division if both operands are integers, and floating-point division if either operand is a float.

> 如果你用Python2的话，除法是不一样的。在两边都是整形的时候，常规除法运算符/就会进行地板除法，而两边只要有一侧是浮点数就会进行浮点除法。

5.2 Boolean expressions 布尔表达式

A boolean expression is an expression that is either true or false. The following examples use the operator ==, which compares two operands and produces True if they are equal and False otherwise:

> 布尔表达式是一种非对即错的表达式，只有这么两个值，true（真）或者false（假）。下面的例子都用了双等号运算符，这个运算符会判断两边的值是否相等，相等就是True，不相等就是False：

```
>>> 5 == 5
>>> 5 == 5
True
>>> 5 == 6
>>> 5 == 6
False
```

True and False are special values that belong to the type bool; they are not strings:

> True和False都是特殊的值，属于bool布尔类型；它们俩不是字符串：

```
>>> type(True)
>>> type(True)
<class 'bool'>
>>> type(False)
>>> type(False)
<class 'bool'>
```

The == operator is one of the relational operators; the others are:

> 双等号运算符是关系运算符的一种，其他关系运算符如下：

```
x != y    # x is not equal to y          二者相等
x > y    # x is greater than y          前者更大
x > y    # x is greater than y          前者更大
x < y    # x is less than y          前者更小
x >= y    # x is greater than or equal to y 大于等于
x >= y    # x is greater than or equal to y 大于等于
x <= y    # x is less than or equal to y     小于等于
```

Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols. A common error is to use a single equal sign (=) instead of a double equal sign (==). Remember that = is an assignment operator and == is a relational operator. There is no such thing as =< or =>.

> 虽然这些运算符你可能很熟悉了，但一定要注意Python里面的符号和数学上的符号有一定区别。常见的错误就是混淆了等号=和双等号==。一定要记住单等号=是一个赋值运算符，而双等号==是关系运算符。另外要注意就是大于等于或者小于等于都是等号放到大于号或者小于号的后面，顺序别弄反。

5.3 Logical operators 逻辑运算符

There are three logical operators: and, or, and not. The semantics (meaning) of these operators is similar to their meaning in English. For example, x > 0 and x < 10 is true only if x is greater than 0 and less than 10.

> 逻辑运算符有三种：且，或以及非。这三种运算符的意思和字面意思差不多。比如x>0且x<10，仅当x在0到10之间的时候才为真。

n%2 == 0 or n%3 == 0 is true if either or both of the conditions is true, that is, if the number is divisible by 2 or 3.

> n%2 == 0 或 n%3 == 0，只要条件有一个成立就是真，就是说这个可以被2或3整除就行了。

Finally, the not operator negates a boolean expression, so not (x > y) is true if x > y is false, that is, if x is less than or equal to y.

> 最后说这个非运算，是针对布尔表达式的，非（x>y）为真，那么x>y就是假的，意味着x小于等于y。

Strictly speaking, the operands of the logical operators should be boolean expressions, but Python is not very strict. Any nonzero number is interpreted as True:

> 严格来说，逻辑运算符的运算对象应该必须是布尔表达式，不过Python就不太严格。任何非零变量都会被认为是真：

```
>>> 42 and True
>>> 42 and True
True
```

This flexibility can be useful, but there are some subtleties to it that might be confusing. You might want to avoid it (unless you know what you are doing).

> 这种灵活性特别有用，不过有的情况下也容易引起混淆。建议你尽量不要这样用，除非你很熟练了。

5.4 Conditional execution 条件执行

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly.Conditional statements give us this ability. The simplest form is the if statement:

> 有用的程序必然要有条件检查判断的功能，根据不同条件要让程序有相应的行为。条件语句就让咱们能够实现这种判断。最简单的就是if语句了：

```
if x > 0:
    print('x is positive')
```

The boolean expression after if is called the condition. If it is true, the indented statement runs. If not, nothing happens.

> if后面的布尔表达式就叫做条件。如果条件为真，随后缩进的语句就运行。如果条件为假，就不运行。

if statements have the same structure as function definitions: a header followed by an indented body. Statements like this are called compound statements.

> if语句与函数定义的结构基本一样：一个头部，后面跟着缩进的语句。这样的语句叫做复合语句。

There is no limit on the number of statements that can appear in the body, but there has to be at least one. Occasionally, it is useful to have a body with no statements (usually as a place keeper for code you haven't written yet). In that case, you can use the pass statement, which does nothing.

> 复合语句中语句体内的语句数量是不限制的，但至少要有一个。有的时候会遇到一个语句体内不放语句的情况，比如空出来用来后续补充。这种情况下，你就可以用pass语句，就是啥也不会做的。

```
if x < 0:
    pass          # TODO: need to handle negative values!
```

5.5 Alternative execution 选择执行

A second form of the if statement is "alternative execution", in which there are two possibilities and the condition determines which one runs. The syntax looks like this:

> if语句的第二种形式就是『选择执行』，这种情况下会存在两种备选的语句，根据条件来判断执行哪一个。语法如下所示：

```
if x % 2 == 0:
    print('x is even')
else:
    print('x is odd')
```

If the remainder when x is divided by 2 is 0, then we know that x is even, and the program displays an appropriate message. If the condition is false, the second set of statements runs. Since the condition must be true or false, exactly one of the alternatives will run. The alternatives are called branches, because they are branches in the flow of execution.

> 如果x除以2的余数为0，x就是一个偶数了，程序就会显示对应的信息。如果条件不成立，那就运行第二条语句。这里条件非真即假，只有两个选择。这些选择也叫『分支』，因为在运行流程上产生了不同的分支。

5.6 Chained conditionals 链式条件

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a chained conditional:

> 有时我们要面对的可能性不只有两种，需要更多的分支。这时候可以使用连锁条件来实现：

```
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

elif is an abbreviation of “else if”. Again, exactly one branch will run. There is no limit on the number of elif statements. If there is an else clause, it has to be at the end, but there doesn’t have to be one.

> elif是『else if』的缩写。这回也还是只会有一个分支的语句会被运行。elif语句的数量是无限制的。如果有else语句的话，这个else语句必须放到整个条件链的末尾，不过else语句并不是必须有的。

```
if choice == 'a':
    draw_a()
elif choice == 'b':
    draw_b()
elif choice == 'c':
    draw_c()
```

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch runs and the statement ends. Even if more than one condition is true, only the first true branch runs.

> 每一个条件都会依次被检查。如果第一个是假，下一个就会被检查，依此类推。如果有一个为真了，相应的分支语句就运行了，这些条件判断的语句就都结束了。如果有一个以上的条件为真，只有先出现的为真的条件所对应的分支语句会运行。

5.7 Nested conditionals 嵌套条件

One conditional can also be nested within another. We could have written the example in the previous section like this:

> 一个条件判断也可以嵌套在另一个条件判断内。上一节的例子可以改写成如下：

```
if x == y:
    print('x and y are equal')
else:
    if x < y:
        print('x is less than y')
    else:
        print('x is greater than y')
```

The outer conditional contains two branches. The first branch contains a simple statement. The second branch contains another if statement, which has two branches of its own. Those two branches are both simple statements, although they could have been conditional statements as well.

> 外部的条件判断包含两个分支。第一个分支只有一个简单的语句。第二个分支包含了另外一重条件判断，这个内部条件判断有两个分支。这两个分支都是简单的语句，他们的位置也可以继续放条件判断语句的。

Although the indentation of the statements makes the structure apparent,nested conditionals become difficult to read very quickly. It is a good idea to avoid them when you can.

> 虽然语句的缩进会让代码结构看着比较清晰明显，但嵌套的条件语句读起来还是有点难度。所以建议你如果可以的话，尽量避免使用嵌套的条件判断。

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

> 逻辑运算符有时候对简化嵌套条件判断很有用。比如下面这个代码就能改写成更简单的版本:

```
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

The print statement runs only if we make it past both conditionals, so we can get the same effect with the and operator:

> 上面的例子中，只有两个条件都满足了才会运行print语句，所以就用逻辑运算符来实现同样的效果即可：

```
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

For this kind of condition, Python provides a more concise option:

> 这种条件下，Python提供了更简洁的表达方法：

```
if 0 < x < 10:
    print('x is a positive single-digit number.')
```

（译者注：Python的这种友善度远远超过了C和C++，这也是为何我一直建议国内高校用Python取代C++来给本科生和研究生做编程入门课程。）

5.8 Recursion 递归运算

It is legal for one function to call another; it is also legal for a function to call itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical things a program can do. For example, look at the following function:

> 一个函数可以去调用另一个函数；函数来调用自己也是允许的。这就是递归，是程序最神奇的功能之一，现在可能还不好理解为什么，那么来看看下面这个函数为例：

```
def countdown(n):
    if n <= 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n-1)
```

If n is 0 or negative, it outputs the word, “Blastoff!” Otherwise, it outputs nand then calls a function named countdown—itself—passing n-1 as an argument. What happens if we call this function like this?

> 如果n为0或者负数，程序会输出『Blastoff！』。其他情况下，程序会调用自身来运行，以自身参数n减去1为参数。如果像下面这样调用这个函数会怎么样？

```
>>> countdown(3)
>>> countdown(3)
```

The execution of countdown begins with n=3, and since n is greater than 0, it outputs the value 3, and then calls itself... The execution of countdown begins with n=2, and since n is greater than 0, it outputs the value 2, and then calls itself... The execution of countdown begins with n=1, and since n is greater than 0, it outputs the value 1, and then calls itself... The execution of countdown begins with n=0, and since n is not greater than 0, it outputs the word, “Blastoff!” and then returns. The countdown that got n=1 returns. The countdown that got n=2 returns. The countdown that got n=3 returns. And then you’re back in __main__. So, the total output looks like this:

> 开始时候函数参数n是3，大于0，输出n的值3，然后调用自身，用n-1也就是2作为参数。。。

接下来的函数参数n是2，大于0，输出n的值2，然后调用自身，用n-1也就是1作为参数。。。再往下去函数参数n是1，大于0，输出n的值1，然后调用自身，用n-1也就是0作为参数。。。最后这次函数参数n是0，等于0了，输出『Blastoff！』，然后返回。 n=1的时候的countdown也执行完了，返回。 n=2的时候的countdown也执行完了，返回。 n=3的时候的countdown也执行完了，返回。 （译者注：这时候一定要注意不是输出字符串就完毕了，要返回的每一个层次的函数调用者。这里不理解的话说明对函数调用的过程掌握的不透彻，一定要好好想仔细了。） 接下来你就回到主函数__main__里面了。所以总的输出会如下所示：

```
3
2
1
Blastoff!
```

A function that calls itself is recursive; the process of executing it is called recursion.

> 调用自身的函数就是递归的；执行这种函数的过程就叫递归运算。

As another example, we can write a function that prints a string n times.

> 我们再写一个用print把一个字符串s显示n次的例子：

```
def print_n(s, n):
    if n <= 0:
        return
    print(s)
    print_n(s, n-1)
s="Python is good"
n=4
print_n(s, n)
```

If n <= 0 the return statement exits the function. The flow of execution immediately returns to the caller, and the remaining lines of the function don't run.

> 如果n小于等于0了，返回语句return就会终止函数的运行。运行流程立即返回到函数调用者，函数其余各行的代码也都不会执行。

The rest of the function is similar to countdown: it displays s and then calls itself to display s n−1 additional times. So the number of lines of output is 1 + (n - 1), which adds up to n.

> 函数其余部分的代码很容易理解：print一下s，然后调用自身，用n-1做参数来继续运行，这样就额外对s进行了n-1次的显示。所以输出的行数是1+（n-1），最终一共有n行输出。

For simple examples like this, it is probably easier to use a for loop. But we will see examples later that are hard to write with a for loop and easy to write with recursion, so it is good to start early.

> 上面这种简单的例子，实际上用for循环更简单。不过后面我们就会遇到一些用for循环不太好写的例子了，这些情况往往用递归更简单，所以早点学习下递归是有好处的。

5.9 Stack diagrams for recursive functions 递归函数的栈图

In Section 3.9, we used a stack diagram to represent the state of a program during a function call. The same kind of diagram can help interpret a recursive function.

> 在本书的第三章第九节，我们用栈图来表征函数调用过程中程序的状态。同样是这种栈图，将有助于给大家展示递归函数的运行过程。

Every time a function gets called, Python creates a frame to contain the function’s local variables and parameters. For a recursive function, there might be more than one frame on the stack at the same time. Figure 5.1 shows a stack diagram for countdown called with n = 3.

> 每次有一个函数被调用的时候，Python都会创建一个框架来包含这个函数的局部变量和形式参数。对于递归函数来说，可能会在栈中同时生成多层次的框架。
>
> 图5.1展示了前面样例中coundown函数在n=3的时候的栈图。

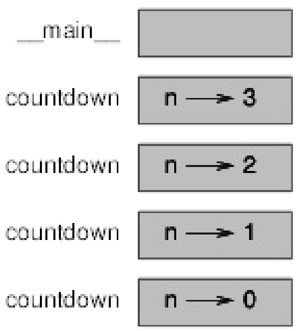


Figure 5.1: Stack diagram.

As usual, the top of the stack is the frame for __main__. It is empty because we did not create any variables in __main__ or pass any arguments to it.

> 栈图的开头依然是主函数__main__。这里主函数是空的，因为我们没有在主函数里面创建变量或者传递参数进去。

The four countdown frames have different values for the parameter n. The bottom of the stack, where n=0, is called the base case. It does not make a recursive call, so there are no more frames.

> 四个coundown方框中形式参数n的值都是不同的。在栈图底部是n=0的时候，也叫基准条件。这时候不再进行递归调用，也就没有更多框架了。

As an exercise, draw a stack diagram for print_n called with s = 'Hello' and n=2. Then write a function called do_n that takes a function object and a number, n, as arguments, and that calls the given function n times.

> 下面练习一下，画一个print_n函数的栈图，让s为字符串『Hello』，n为2。然后写一个函数，名字为do_n，使用一个操作对象和一个数字n作为实际参数，给出一个n作为次数来调用这个函数。

5.10 Infinite recursion 无穷递归

If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates. This is known as infinite recursion, and it is generally not a good idea. Here is a minimal program with an infinite recursion:

> 如果一个递归一直都不能到达基准条件，那就会持续不断地进行自我调用，程序也就永远不会终止了。这就叫无穷递归，一般这都不是个好事情哈。下面就是一个无穷递归的最简单的例子：

```
def recurse():
    recurse()
```

In most programming environments, a program with infinite recursion does not really run forever. Python reports an error message when the maximum recursion depth is reached:

> 在大多数的开发环境下，无穷递归的程序并不会真的永远运行下去。Python会在函数达到允许递归的最大层次后返回一个错误信息：

```
File "<stdin>", line 2, in recurse
File "<stdin>", line 2, in recurse
File "<stdin>", line 2, in recurse
File "<stdin>", line 2, in recurse
RuntimeError: Maximum recursion depth exceeded
```

This traceback is a little bigger than the one we saw in the previous chapter. When the error occurs, there are 1000 recurse frames on the stack!

> 这个追踪会我们之前看到的长很多。这种错误出现的时候，栈中都已经有1000层递归框架了！

If you write encounter an infinite recursion by accident, review your function to confirm that there is a base case that does not make a recursive call. And if there is a base case, check whether you are guaranteed to reach it.

> 如果你意外写出来一个无穷递归的代码，好好检查一下你的函数，一定要确保有一个基准条件来停止递归调用。如果存在了基准条件，检查一下一定要确保能使之成立。

5.11 Keyboard input 键盘输入

The programs we have written so far accept no input from the user. They just do the same thing every time.

> 目前为止咱们写过的程序还都没有接收过用户的输入。这写程序每次都是做一些同样的事情。

Python provides a built-in function called input that stops the program and waits for the user to type something. When the user presses Return or Enter, the program resumes and input returns what the user typed as a string. In Python 2, the same function is called raw_input.

> Python提供了内置的一个函数，名叫input，这个函数会停止程序运行，等待用户来输入一些内容。用户按下ESC或者Enter回车键，程序就恢复运行，input函数就把用户输入的内容作为字符串返回。在Python2里面，同样的函数名字不同，叫做raw_input。

```
>>> text = input()
>>> text = input()
What are you waiting for?
>>> text
>>> text
What are you waiting for?
```

Before getting input from the user, it is a good idea to print a prompt telling the user what to type. input can take a prompt as an argument:

> 在用户输入内容之前，最好显示一些提示，来告诉用户需要输入什么内容。input函数能够把提示内容作为参数：

```
>>> name = input('What...is your name?\n')
>>> name = input('What...is your name?\n')
What...is your name?
Arthur, King of the Britons!
>>> name
>>> name
Arthur, King of the Britons!
```

The sequence \n at the end of the prompt represents a newline, which is a special character that causes a line break. That's why the user's input appears below the prompt.

> 提示内容末尾的\n表示要新建一行，这是一个特殊的字符，表示换行。因为有了换行字符，所以用户输入就跑到了提示内容下面去了。

If you expect the user to type an integer, you can try to convert the return value to int:

> 如果你想要用户来输入一个整形变量，可以把返回的值手动转换一下：

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
>>> speed = input(prompt)
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
42
>>> int(speed)
>>> int(speed)
42
```

But if the user types something other than a string of digits, you get an error:

> 如果用户输入的是其他内容，而不是一串数字，就会得到一个错误了：

```
>>> speed = input(prompt)
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed) ValueError: invalid literal for int() with base 10
>>> int(speed) ValueError: invalid literal for int() with base 10
```

We will see how to handle this kind of error later.

> 稍后我们再来看看如何应对这种错误。

5.12 Debugging 调试

When a syntax or runtime error occurs, the error message contains a lot of information, but it can be overwhelming. The most useful parts are usually:

> 当语法错误或者运行错误出现的时候，错误信息会包含很多有用的信息，不过信息量太大，太繁杂。最有用的也就下面这两类：

- What kind of error it was, and

> 错误的类型是什么，以及

- Where it occurred.

> 错误的位置在哪里。

Syntax errors are usually easy to find, but there are a few gotchas. Whitespace errors can be tricky because spaces and tabs are invisible and we are used to ignoring them.

```
>>> x = 5
>>> x = 5
>>>  y = 6
>>>  y = 6
    File "<stdin>", line 1
    y = 6
    ^
IndentationError: unexpected indent
```

In this example, the problem is that the second line is indented by one space. But the error message points to y, which is misleading. In general, error messages indicate where the problem was discovered, but the actual error might be earlier in the code, sometimes on a previous line.

> 这个例子里面，错误的地方是第二行开头用一个空格来缩进了。但这个错误是指向y的，这就有点误导了。一般情况下，错误信息都会表示出发现问题的位置，但具体的错误可能是在此位置之前的代码引起的，有的时候甚至是前一行。

The same is true of runtime errors. Suppose you are trying to compute a signal-to-noise ratio in decibels. The formula is $SNR_{db} = 10 \log_{10}(P_{signal}/P_{noise})$ In Python, you might write something like this:

> 同样情况也发生在运行错误的情况下。假设你试着用分贝为单位来计算信噪比。

公式为： $SNR_{db} = 10 \log_{10}(P_{signal}/P_{noise})$

> 在Python，你可能像下面这样写：

```
import math
signal_power = 9
noise_power = 10
ratio = signal_power // noise_power
decibels = 10 * math.log10(ratio) print(decibels)
```

When you run this program, you get an exception:

> 运行这个程序，你就会得到如下错误信息：

```
Traceback (most recent call last):
    File "snr.py", line 5, in ?
        decibels = 10 * math.log10(ratio)
ValueError: math domain error
```

The error message indicates line 5, but there is nothing wrong with that line. To find the real error, it might be useful to print the value of ratio, which turns out to be 0. The problem is in line 4, which uses floor division instead of floating-point division.

> 这个错误信息提示第五行，但那一行实际上并没有错。要找到真正的错误，就要输出一下ratio的值来看一下，结果发现是0了。那问题实际是在第四行，应该用浮点除法，结果多打了一个右斜杠，弄成了地板除法，才导致的错误。

You should take the time to read error messages carefully, but don’t assume that everything they say is correct.

> 所以你得花点时间仔细阅读错误信息，但不要轻易就认为出错信息说的内容都是完全正确可靠的。

5.13 Glossary 术语列表

floor division: An operator, denoted //, that divides two numbers and rounds down (toward zero) to an integer.

> 地板除法：一种运算符，双右斜杠，把两个数相除，舍弃小数位，结果为整形。

modulus operator: An operator, denoted with a percent sign (%), that works on integers and returns the remainder when one number is divided by another.

> 求模取余：一种运算符，百分号%，对整形起作用，返回两个数字相除的余数。

boolean expression: An expression whose value is either True or False.

> 布尔表达式：一种值为真或假的表达式。

relational operator: One of the operators that compares its operands: ==, !=, >, <, >=, and <=.

> 关系运算符：对比运算对象关系的运算符：==相等, !=不等, >大于, <小于, >=大于等于, 以及<=小于等于。

logical operator: One of the operators that combines boolean expressions: and, or, and not.

> 逻辑运算符：把布尔表达式连接起来的运算符：and且，or或，以及not非。

conditional statement: A statement that controls the flow of execution depending on some condition.

> 条件语句：控制运行流程的语句，根据不同条件有不同语句去运行。

condition: The boolean expression in a conditional statement that determines which branch runs.

> 条件：条件语句所适用的布尔表达式，根据真假来决定运行分支。

compound statement: A statement that consists of a header and a body. The header ends with a colon (:). The body is indented relative to the header.

> 复合语句：包含头部与语句体的一套语句组合。头部要有冒号做结尾，语句体相对于头部要有一次缩进。

branch: One of the alternative sequences of statements in a conditional statement.

> 分支：条件语句当中备选的一系列语句。

chained conditional: A conditional statement with a series of alternative branches.

> 链式条件：一系列可选分支构成的条件语句。

nested conditional: A conditional statement that appears in one of the branches of another conditional statement.

> 嵌套条件：条件语句分支中继续包含次级条件语句的情况。

return statement: A statement that causes a function to end immediately and return to the caller.

> 返回语句：一种特殊的语句，功能是终止当前函数，立即跳出到函数调用者。

recursion: The process of calling the function that is currently executing.

> 递归：函数对自身进行调用的过程。

base case: A conditional branch in a recursive function that does not make a recursive call.

> 基准条件：递归函数中一个条件分支，要实现终止递归调用。

infinite recursion: A recursion that doesn't have a base case, or never reaches it. Eventually, an infinite recursion causes a runtime error.

> 无穷递归：一个没有基准条件的递归，或者永远无法达到基准条件的递归。一般无穷递归总会引起运行错误。

5.14 Exercises 练习

Exercise 1 练习1

The time module provides a function, also named time, that returns the current Greenwich Mean Time in "the epoch", which is an arbitrary time used as a reference point. On UNIX systems, the epoch is 1 January 1970.

> time模块提供了一个名字同样叫做time的函数，会返回当前格林威治时间的时间戳，就是以某一个时间点作为初始参考值。在Unix系统中，时间戳的参考值是1970年1月1号。
>
> （译者注：时间戳就是系统当前时间相对于1970.1.1 00:00:00以秒计算的偏移量，时间戳是惟一的。）

```
>>> import time
>>> import time
>>> time.time() 1437746094.5735958
>>> time.time() 1437746094.5735958
```

Write a script that reads the current time and converts it to a time of day in hours, minutes, and seconds, plus the number of days since the epoch.

> 写一个脚本，读取当前的时间，把这个时间转换以天为单位，剩余部分转换成小时-分钟-秒的形式，加上参考时间以来的天数。

Exercise 2 练习2

Fermat's Last Theorem says that there are no positive integers a, b, and c such that $a^n + b^n = c^n$ for any values of n greater than 2.

> 费马大定理内容为，a、b、c、n均为正整数，在n大于2的情况，下面的等式关系不成立：
>
> $$a^n + b^n = c^n$$

1. Write a function named check_fermat that takes four parameters—a, b, c and n—and checks to see if Fermat's theorem holds. If n is greater than 2 and

> 写一个函数，名叫check_fermat，这个函数有四个形式参数：a、b、c以及n，检查一下费马大定理是否成立，看看在n大于2的情况下下列等式
>
> $$a^n + b^n = c^n$$
>
> 是否成立。

1. The program should print, "Holy smokes, Fermat was wrong!" Otherwise the program should print, "No, that doesn't work."

> 要求程序输出『Holy smokes, Fermat was wrong!』或者『No, that doesn't work.』

1. Write a function that prompts the user to input values for a, b, c and n, converts them to integers, and uses check_fermat to check whether they violate Fermat's theorem.

> 写一个函数来提醒用户要输入a、b、c和n的值，然后把输入值转换为整形变量，接着用check_fermat这个函数来检查他们是否违背了费马大定理。

Exercise 3 练习3

If you are given three sticks, you may or may not be able to arrange them in a triangle. For example, if one of the sticks is 12 inches long and the other two are one inch long, you will not be able to get the short sticks to meet in the middle. For any three lengths, there is a simple test to see if it is possible to form a triangle:

> 给你三根木棍，你能不能把它们拼成三角形呢？比如一个木棍是12英寸长，另外两个是1英寸长，这两根短的就不够长，无法拼成三角形了。
>
> （译者注：1英寸=2.54厘米）对于任意的三个长度，有一个简单的方法来检测它们能否拼成三角形：

If any of the three lengths is greater than the sum of the other two, then you cannot form a triangle. Otherwise, you can. (If the sum of two lengths equals the third, they form what is called a “degenerate” triangle.)

> 只要三个木棍中有任意一个的长度大于其他两个的和，就拼不成三角形了。必须要任意一个长度都小于两边和才能拼成三角形。（如果两边长等于第三边，就只能组成所谓『退化三角形』了。译者注：实际上这不就成了线段了么？）

1. Write a function named is_triangle that takes three integers as arguments, and that prints either “Yes” or “No”, depending on whether you can or cannot form a triangle from sticks with the given lengths.

> 写一个叫做is_triangle的函数，用三个整形变量为实际参数，函数根据你输入的值能否拼成三角形来判断输出『Yes』或者『No』。

1. Write a function that prompts the user to input three stick lengths, converts them to integers, and uses is_triangle to check whether sticks with the given lengths can form a triangle.

> 写一个函数来提示下用户，要输入三遍长度，把它们转换成整形，用is_triangle函数来检测这些给定长度的边能否组成三角形。

Exercise 4 练习4

What is the output of the following program? Draw a stack diagram that shows the state of the program when it prints the result.

> 下面的代码输出会是什么？画一个栈图来表示一下如下例子中程序输出结果时候的状态。

```
def recurse(n, s):
    if n == 0:
        print(s)
    else:
        recurse(n-1, n+s)
recurse(3, 0)
```

1. What would happen if you called this function like this: recurse(-1, 0)?

> recurse(-1, 0)这样的调用函数会有什么效果？

1. Write a docstring that explains everything someone would need to know in order to use this function (and nothing else).

> 为这个函数写一个文档字符串，解释一下用法（仅此而已）。

The following exercises use the turtle module, described in Chapter 4:

> 接下来的练习用到了第四章我们提到过的turtle小乌龟模块。

Exercise 5 练习5

Read the following function and see if you can figure out what it does. Then run it (see the examples in Chapter 4).

> 阅读下面的函数，看看你能否弄清楚函数的作用。运行一下试试（参考第四章里面的例子来酌情修改代码）。

```
def draw(t, length, n):
    if n == 0:
        return
    angle = 50
    t.fd(length*n)
    t.lt(angle)
    draw(t, length, n-1)
    t.rt(2*angle)
    draw(t, length, n-1)
    t.lt(angle)
    t.bk(length*n)
```

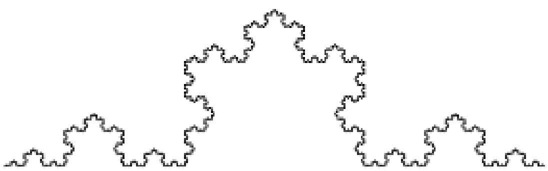


Figure 5.2: A Koch curve.

Exercise 6 练习6

The Koch curve is a fractal that looks something like Figure 5.2. To draw a Koch curve with length x, all you have to do is

> Koch科赫曲线是一种分形曲线，外观如图5.2所示。要画长度为x的这种曲线，你要做的步骤如下：

1. Draw a Koch curve with length x/3.

> 画一个长度为三分之一x的Koch曲线。

1. Turn left 60 degrees.

> 左转60度。

1. Draw a Koch curve with length x/3.

> 画一个长度为三分之一x的Koch曲线。

1. Turn right 120 degrees.

> 右转120度。

1. Draw a Koch curve with length x/3.

> 画一个长度为三分之一x的Koch曲线。

1. Turn left 60 degrees.

> 左转60度。

1. Draw a Koch curve with length x/3.

> 画一个长度为三分之一x的Koch曲线。

The exception is if x is less than 3: in that case, you can just draw a straight line with length x.

> 特例是当x小于3的时候：这种情况下，你就可以只画一个长度为x的直线段。

1. Write a function called koch that takes a turtle and a length as parameters, and that uses the turtle to draw a Koch curve with the given length.

> 写一个叫做koch的函数，用一个小乌龟turtle以及一个长度length做形式参数，用这个小乌龟来画给定长度length的Koch曲线。

1. Write a function called snowflake that draws three Koch curves to make the outline of a snowflake.Solution.

> 写一个叫做snowflake的函数，画三个Koch曲线来制作一个雪花的轮廓。参考代码

1. The Koch curve can be generalized in several ways. See here for examples and implement your favorite.

> 生成Koch曲线的方法还有很多。点击 这里来查看更多的例子，探索一下看看你喜欢哪个。

Chapter 6 Fruitful functions 有返回值的函数

Many of the Python functions we have used, such as the math functions, produce return values. But the functions we've written are all void: they have an effect, like printing a value or moving a turtle, but they don't have a return value. In this chapter you will learn to write fruitful functions.

> 我们已经用过的很多Python的函数，比如数学函数，都会有返回值。但我们写过的函数都是无返回值的：他们实现一些效果，比如输出一些值，或者移动小乌龟，但他们就是不返回值。

6.1 Return values 返回值

Calling the function generates a return value, which we usually assign to a variable or use as part of an expression.

> 对函数进行调用，就会产生一个返回的值，我们一般把这个值赋给某个变量，或者放进表达式中来用。

```
e = math.exp(1.0) height = radius * math.sin(radians)
```

The functions we have written so far are void. Speaking casually, they have no return value; more precisely, their return value is None.

> 目前为止，我们写过的函数都没有返回值。简单说是没有返回值，更确切的讲，这些函数的返回值是空（None）。

In this chapter, we are (finally) going to write fruitful functions. The first example is area, which returns the area of a circle with the given radius:

> 在本章，我们总算要写一些有返回值的函数了。第一个例子就是一个计算给定半径的圆的面积的函数：

```
def area(radius):
    a = math.pi * radius**2
    return a
```

We have seen the return statement before, but in a fruitful function the return statement includes an expression. This statement means: "Return immediately from this function and use the following expression as a return value." The expression can be arbitrarily complicated, so we could have written this function more concisely:

> 返回语句我们之前已经遇到过了，但在有返回值的函数里面，返回语句可以包含表达式。这个返回语句的意思是：立即返回下面这个表达式作为返回值。返回语句里面的表达式可以随便多复杂都行，所以刚刚那个计算面积的函数我们可以精简改写成以下形式：

```
def area(radius):
    return math.pi * radius**2
```

On the other hand, temporary variables like a can make debugging easier. Sometimes it is useful to have multiple return statements, one in each branch of a conditional:

> 另外，有一些临时变量可以让后续的调试过程更简单。所以有时候可以多设置几条返回语句，每一条都对应一种情况。

```
def absolute_value(x):
    if x < 0:
        return -x
    else:
        return x
```

Since these return statements are in an alternative conditional, only one runs.

> 因为这些返回语句对应的是不同条件，因此实际上最终只会有一个返回动作执行。

As soon as a return statement runs, the function terminates without executing any subsequent statements. Code that appears after a return statement, or any other place the flow of execution can never reach, is called dead code. In a fruitful function, it is a good idea to ensure that every possible path through the program hits a return statement. For example:

> 返回语句运行的时候，函数就结束了，也不会运行任何其他的语句了。返回语句后面的代码，执行流程里所有其他的位置都无法再触碰了，这种代码叫做『死亡代码』。在有返回值的函数里面，建议要确认一下每一种存在的可能，来让函数触发一个返回语句。下面例子中：

```
def absolute_value(x):
    if x < 0:
        return -x
    if x > 0:
        return x
```

This function is incorrect because if x happens to be 0, neither condition is true, and the function ends without hitting a return statement. If the flow of execution gets to the end of a function, the return value is None, which is not the absolute value of 0.

> 这个函数就是错误的，因为一旦x等于0了，两个条件都没满足，没有触发返回语句，函数就结束了。执行流程走完这个函数之后，返回的就是空（None），而本应该返回0的绝对值的。

```
>>> absolute_value(0)
>>> absolute_value(0)
None
```

By the way, Python provides a built-in function called abs that computes absolute values. As an exercise, write a compare function takes two values, x and y, and returns 1 if x > y, 0 if x == y, and -1 if x < y.

> 顺便说一下，Python内置函数就有一个叫abs的，就是用来求绝对值的。
>
> 然后练习一下把，写一个比较大小的函数，用两个之x和y作为参数，如果x大于y返回1，相等返回0，x小于y返回-1.

6.2 Incremental development 增量式开发

As you write larger functions, you might find yourself spending more time debugging. To deal with increasingly complex programs, you might want to try a process called incremental development. The goal of incremental development is to avoid long debugging sessions by adding and testing only a small amount of code at a time. As an example, suppose you want to find the distance between two points, given by the coordinates (x1, y1) and (x2, y2). By the Pythagorean theorem, the distance is:

> 写一些复杂函数的时候，你会发现要花很多时间调试。
>
> 要应对越来越复杂的程序，你不妨来试试增量式开发的办法。增量式开发的目的是避免长时间的调试过程，一点点对已有的小规模代码进行增补和测试。

```
distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}
```

The first step is to consider what a distance function should look like in Python. In other words, what are the inputs (parameters) and what is the output (return value)?

> 首先大家来想一下用Python来计算两点距离的函数应该是什么样。换句话说，输入的参数是什么，输出的返回值是什么？

In this case, the inputs are two points, which you can represent using four numbers. The return value is the distance represented by a floating-point value. Immediately you can write an outline of the function:

> 这个案例里面，输入的应该是两个点的坐标，平面上就是四个数字了。返回的值是两点间的距离，就是一个浮点数了。

```
def distance(x1, y1, x2, y2):
    return 0.0
```

Obviously, this version doesn’t compute distances; it always returns zero. But it is syntactically correct, and it runs, which means that you can test it before you make it more complicated. To test the new function, call it with sample arguments:

> 当然了，上面这个版本的代码肯定算不出距离了；不管输入什么都会返回0了。但这个函数语法上正确，而且可以运行，这样在程序过于复杂的情况之前就能及时测试了。
>
> 要测试一下这个新函数，就用简单的参数来调用一下吧：

```
>>> distance(1, 2, 4, 6)
>>> distance(1, 2, 4, 6)
0.0
```

I chose these values so that the horizontal distance is 3 and the vertical distance is 4; that way, the result is 5, the hypotenuse of a 3-4-5 triangle. When testing a function, it is useful to know the right answer.

> 我选择这些数值，水平的距离就是3，竖直距离就是4；这样的话，平面距离就应该是5了，是一个3-4-5三角形的斜边长了。我们已经知道正确结果应该是什么了，这样对测试来说很有帮助。

At this point we have confirmed that the function is syntactically correct, and we can start adding code to the body. A reasonable next step is to find the differences x2 − x1 and y2 − y1. The next version stores those values in temporary variables and prints them.

> 现在我们已经确认过了，这个函数在语法上是正确的，接下来我们就可以在函数体内添加代码了。下一步先添加一下求x2-x1和y2-y1的值的内容。接下来的版本里面，就把它们存在一些临时变量里面，然后输出一下。

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print('dx is', dx)
    print('dy is', dy)
    return 0.0
```

If the function is working, it should display 'dx is 3' and 'dy is 4'. If so, we know that the function is getting the right arguments and performing the first computation correctly. If not, there are only a few lines to check. Next we compute the sum of squares of dx and dy:

> 这个函数如果工作的话，应该显示出'dx is 3'和'dy is 4'。如果成功显示了，我们就知道函数已经得到了正确的实际参数，并且正确进行了初步的运算。如果没有显示，只要检查一下这么几行代码就可以了。接下来，就要计算dx和dy的平方和了。

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    print('dsquared is: ', dsquared)
    return 0.0
```

Again, you would run the program at this stage and check the output (which should be 25). Finally, you can use math.sqrt to compute and return the result:

> 在这一步，咱们依然亏运行一下程序，来检查输出，输出的应该是25。输出正确的话，最后一步就是用math.sqrt这个函数来计算并返回结果：

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    return result
```

If that works correctly, you are done. Otherwise, you might want to print the value of result before the return statement.

> 如果程序工作没问题，就搞定了。否则你可能就需要用print输出一下最终计算结果，然后再返回这个值。

The final version of the function doesn't display anything when it runs; it only returns a value. The print statements we wrote are useful for debugging, but once you get the function working, you should remove them. Code like that is called scaffolding because it is helpful for building the program but is not part of the final product.

> 此函数的最终版本在运行的时候是不需要显示任何内容的；这个函数只需要返回一个值。我们写得这些print打印语句都是用来调试的，但一旦程序能正常工作了，就应该把print语句去掉。这些print代码也叫『脚手架代码』因为是用来构建程序的，但不会被存放在最终版本的程序中。

When you start out, you should add only a line or two of code at a time. As you gain more experience, you might find yourself writing and debugging bigger chunks. Either way, incremental development can save you a lot of debugging time. The key aspects of the process are:

> 当你动手的时候，每次建议只添加一两行代码。等你经验更多了，你发现自己可能就能够驾驭大块代码了。不论如何，增量式开发总还是能帮你节省很多调试消耗的时间。
>
> 这个过程的核心如下：

1. Start with a working program and make small incremental changes. At any point, if there is an error, you should have a good idea where it is.

> 一定要用一个能工作的程序来开始，每次逐渐添加一些细小增补。在任何时候遇到错误，都应该弄明白错误的位置。

1. Use variables to hold intermediate values so you can display and check them.

> 用一些变量来存储中间值，这样你可以显示一下这些值，来检查一下。

1. Once the program is working, you might want to remove some of the scaffolding or consolidate multiple statements into compound expressions, but only if it does not make the program difficult to read.

> 程序一旦能工作了，你就应该把一些发挥『脚手架作用』的代码删掉，并且把重复的语句改写成精简版本，但尽量别让程序变得难以阅读。

As an exercise, use incremental development to write a function called hypotenuse that returns the length of the hypotenuse of a right triangle given the lengths of the two legs as arguments. Record each stage of the development process as you go.

> 做个练习吧，用这种增量式开发的思路来写一个叫做hypotenuse（斜边）的函数，接收两个数值作为给定两边长，求以两边长为直角边的直角三角形斜边的长度。做练习的时候记得要记录好开发的各个阶段。

6.3 Composition 组合

As you should expect by now, you can call one function from within another. As an example, we'll write a function that takes two points, the center of the circle and a point on the perimeter, and computes the area of the circle. Assume that the center point is stored in the variables xc and yc, and the perimeter point is in xp and yp. The first step is to find the radius of the circle, which is the distance between the two points. We just wrote a function,distance, that does that:

> 你现在应该已经能够在一个函数里面调用另外一个函数了。下面我们写一个函数作为例子，这个函数需要两个点，一个是圆心，一个是圆周上面的点，函数要用来计算这个圆的面积。
>
> 假设圆心的坐标存成一对变量：xc和yc，圆周上一点存成一对变量：xp和yp。第一步就是算出来这个圆的半径，也就是这两个点之间的距离。我们就用之前写过的那个distance的函数来完成这件事：

```
radius = distance(xc, yc, xp, yp)
```

The next step is to find the area of a circle with that radius; we just wrote that, too:

> 下一步就是根据计算出来的半径来算圆的面积；计算面积的函数我们也写过了：

```
result = area(radius)
```

Encapsulating these steps in a function, we get:

> 把上述的步骤组合在一个函数里面：

```
def circle_area(xc, yc, xp, yp):
    radius = distance(xc, yc, xp, yp)
    result = area(radius)
    return result
```

The temporary variables radius and result are useful for development and debugging, but once the program is working, we can make it more concise by composing the function calls:

> 临时变量radius和result是用于开发和调试用的，只要程序能正常工作了，就可以把它们都精简下去：

```
def circle_area(xc, yc, xp, yp):
    return area(distance(xc, yc, xp, yp))
```

6.4 Boolean functions 布尔函数

Functions can return booleans, which is often convenient for hiding complicated tests inside functions. For example:

> 函数也可以返回布尔值，这种情况便于隐藏函数内部的复杂测试。例如：

```
def is_divisible(x, y):
    if x % y == 0:
        return True
    else:
        return False
```

It is common to give boolean functions names that sound like yes/no questions; is_divisible returns either True or False to indicate whether x is divisible by y. Here is an example:

> 一般情况下都给这种布尔函数起个独特的名字，比如要有判断意味的提示；is_divisible 这个函数就去判断x能否被y整除而对应地返回真或假。

```
>>> is_divisible(6, 4)
>>> is_divisible(6, 4)
False
>>> is_divisible(6, 3)
>>> is_divisible(6, 3)
True
```

The result of the == operator is a boolean, so we can write the function more concisely by returning it directly:

> 双等号运算符的返回结果是一个布尔值，所以我们可以用下面的方法来简化刚刚的函数：

```
def is_divisible(x, y):
    return x % y == 0
```

Boolean functions are often used in conditional statements:

> 布尔函数经常用于条件语句：

```
if is_divisible(x, y):
    print('x is divisible by y')
```

It might be tempting to write something like:

> 可以用于写如下这种代码：

```
if is_divisible(x, y) == True:
    print('x is divisible by y'
```

But the extra comparison is unnecessary. As an exercise, write a function is_between(x, y, z) that returns True if x ≤ y ≤z or False otherwise.

> 但额外的比较并没有必要。
>
> 做一个练习，写一个函数is_between(x, y, z)，如果x ≤ y ≤z则返回真，其他情况返回假。

6.5 More recursion 更多递归

We have only covered a small subset of Python, but you might be interested to know that this subset is a complete programming language, which means that anything that can be computed can be expressed in this language. Any program ever written could be rewritten using only the language features you have learned so far (actually, you would need a few commands to control devices like the mouse, disks, etc., but that's all).

> 我们目前学过的知识Python的一小部分子集，不过这部分子集本身已经是一套完整的编程语言了，这就意味着所有能计算的东西都可以用这部分子集来表达。实际上任何程序都可以改写成只用你所学到的这部分Python特性的代码。（当然你还需要一些额外的代码来控制设备，比如鼠标、磁盘等等，但也就这么多额外需要而已。）

Proving that claim is a nontrivial exercise first accomplished by Alan Turing, one of the first computer scientists (some would argue that he was a mathematician, but a lot of early computer scientists started as mathematicians). Accordingly, it is known as the Turing Thesis. For a more complete (and accurate) discussion of the Turing Thesis, I recommend Michael Sipser's book Introduction to the Theory of Computation.

> 阿兰图灵最先证明了这个说法，他是最早的计算机科学家之一，有人会认为他更是一个数学家，不过早起的计算机科学家也都是作为数学家来起步的。因此这个观点也被叫做图灵理论。关于对此更全面也更准确的讨论，我推荐一本Michael Sipser的书：Introduction to the Theory of Computation 计算方法导论。

To give you an idea of what you can do with the tools you have learned so far, we'll evaluate a few recursively defined mathematical functions. A recursive definition is similar to a circular definition, in the sense that the definition contains a reference to the thing being defined. A truly circular definition is not very useful:

> 为了让你能更清晰地认识到自己当前学过的这些内容能用来做什么，我们会看看一些数学的函数，这些函数都是递归定义的。递归定义与循环定义有些相似，就是函数的定义体内包含了对所定义内容的引用。一一个完全循环的定义并不会有太大用。

vorpal: An adjective used to describe something that is vorpal.

> 刺穿的：用来描述被刺穿的形容词。

If you saw that definition in the dictionary, you might be annoyed. On the other hand, if you looked up the definition of the factorial function, denoted with the symbol !, you might get something like this:

> 你在词典里面看到上面这种定义，一定很郁闷。然而如果你查一下阶乘函数的定义，你估计会得到如下结果：

```
0! = 1
n! = n (n−1)!
```

This definition says that the factorial of 0 is 1, and the factorial of any other value, n, is n multiplied by the factorial of n−1.

> 这个定义表明了0的阶乘为1，然后对任意一个数n的阶乘，是n与n-1阶乘相乘。

So 3! is 3 times 2!, which is 2 times 1!, which is 1 times 0!. Putting it all together, 3! equals 3 times 2 times 1 times 1, which is 6.

> 所以3的阶乘就是3乘以2的阶乘，而2的阶乘就是2乘以1的阶乘，1的阶乘是1乘以0的阶乘。算到一起，3的阶乘就等于321*1，就是6了。

If you can write a recursive definition of something, you can write a Python program to evaluate it. The first step is to decide what the parameters should be. In this case it should be clear that factorial takes an integer:

> 如果要给某种对象写一个递归的定义，就可以用Python程序来实现。第一步是要来确定形式参数是什么。在这种情况下要明确阶乘所用到的都应该是整形：

```
def factorial(n):
```

If the argument happens to be 0, all we have to do is return 1:

> 如果传来的实际参数碰巧是0，要返回1：

```
def factorial(n):
    if n == 0:
        return 1
```

Otherwise, and this is the interesting part, we have to make a recursive call to find the factorial of n−1 and then multiply it by n:

> 其他情况就有意思了，我们必须用递归的方式来调用n-1的阶乘，然后用它来乘以n：

```
def factorial(n):
    if n == 0:
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
    return result
```

The flow of execution for this program is similar to the flow of countdown in Section 5.8. If we call factorial with the value 3:

> 这个程序的运行流程和5.8里面的那个倒计时有点相似。我们用3作为参数来调用一下这个阶乘函数试试：

Since 3 is not 0, we take the second branch and calculate the factorial of n-1...

> 3不是0，所以分支一下，计算n-1的阶乘。。。

Since 2 is not 0, we take the second branch and calculate the factorial of n-1...

> 2不是0，所以分支一下，计算n-1的阶乘。。。

Since 1 is not 0, we take the second branch and calculate the factorial of n-1...

> 1不是0，所以分支一下，计算n-1的阶乘。。。

Since 0 equals 0, we take the first branch and return 1 without making any more recursive calls.

> 到0了，就返回1给进行递归的分支。

The return value, 1, is multiplied by n, which is 1, and the result is returned.

> 返回值1与1相乘，结果再次返回。

The return value, 1, is multiplied by n, which is 2, and the result is returned.

> 返回值1与2相乘，结果再次返回。

The return value (2) is multiplied by n, which is 3, and the result, 6, becomes the return value of the function call that started the whole process.

> 2的返回值再与n也就是3想成，得到的结果是6，就成了整个流程最终得到的答案。

Figure 6.1 shows what the stack diagram looks like for this sequence of function calls.

> 图6.1表明了这一系列函数调用过程中的栈图。

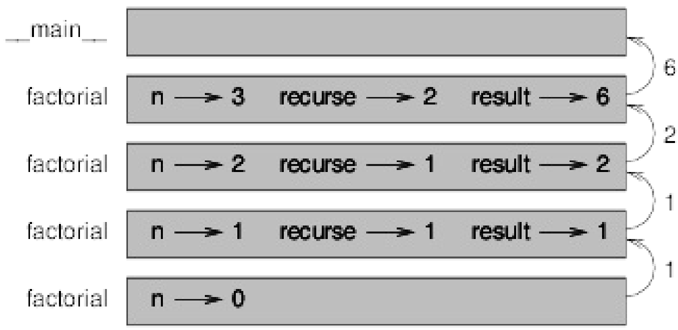


Figure 6.1: Stack diagram.

The return values are shown being passed back up the stack. In each frame, the return value is the value of result, which is the product of n and recurse. In the last frame, the local variables recurse and result do not exist, because the branch that creates them does not run.

6.6 Leap of faith 思维跳跃

Following the flow of execution is one way to read programs, but it can quickly become overwhelming. An alternative is what I call the “leap of faith”. When you come to a function call, instead of following the flow of execution, you assume that the function works correctly and returns the right result.

> 跟随着运行流程是阅读程序的一种方法，但很快就容易弄糊涂。另外一个方法，我称之为『思维跳跃』。当你遇到一个函数调用的时候，你不用去追踪具体的执行流程，而是假设这个函数工作正常并且返回正确的结果。

In fact, you are already practicing this leap of faith when you use built-in functions. When you call math.cos or math.exp, you don’t examine the bodies of those functions. You just assume that they work because the people who wrote the built-in functions were good programmers.

> 实际上你已经联系过这种思维跳跃了，就在你使用内置函数的时候。当你调用math.cos或者math.exp的时候，你并没有仔细查看这些函数的函数体。你就假设他们都工作，因为写这些内置函数的人都是很可靠的编程人员。

The same is true when you call one of your own functions. For example, in Section 6.4, we wrote a function called is_divisible that determines whether one number is divisible by another. Once we have convinced ourselves that this function is correct—by examining the code and testing—we can use the function without looking at the body again.

> 你调用自己写的函数时候也是同样道理。比如在6.4部分，我们谢了这个叫做is_divisible的函数来判断一个数能否被另外一个数整除。一旦我们通过分析代码和做测试来确定了这个函数没有问题，我们就可以直接使用这个函数了，不用去理会函数体内部细节了。

The same is true of recursive programs. When you get to the recursive call, instead of following the flow of execution, you should assume that the recursive call works (returns the correct result) and then ask yourself, “Assuming that I can find the factorial of n−1, can I compute the factorial of n?” It is clear that you can, by multiplying by n.

> 对于递归函数而言也是同样道理。当你进行递归调用的时候，并不用追踪执行流程，你只需要假设递归调用正常工作，返回正确结果，然后你可以问问自己：『假设我能算出来n-1的阶乘，我能否计算出n的阶乘呢？』很显然你是可以的，乘以n就可以了。

Of course, it’s a bit strange to assume that the function works correctly when you haven’t finished writing it, but that’s why it’s called a leap of faith!

> 当然了，当你还没写完一个函数的时候就假设它正常工作确实有点奇怪，不过这也是我们称之为『思维飞跃』的原因了，你总得飞跃一下。

6.7 One more example 斐波拉契数列

After factorial, the most common example of a recursively defined mathematical function is fibonacci, which has the following definition (see Here):

> 计算阶乘之后，我们来看看斐波拉契数列，这是一个广泛应用于展示递归定义的数学函数，[定义](http://en.wikipedia.org/wiki/Fibonacci_number如下：

```
fibonacci(0) = 0
  fibonacci(1) = 1
  fibonacci(n) = fibonacci(n−1) + fibonacci(n−2)
```

Translated into Python, it looks like this:

> 翻译成Python的语言大概如下这样：

```
def fibonacci (n):
    if n == 0:
        return 0
    elif  n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

If you try to follow the flow of execution here, even for fairly small values of n, your head explodes. But according to the leap of faith, if you assume that the two recursive calls work correctly, then it is clear that you get the right result by adding them together.

> 这里如果你还想尝试着追踪执行流程，你一定得累死，即便是一些很小的n，你都得头炸掉了。但根据『思维飞跃』的方法，如果你假设两个递归调用都正常工作，整个过程就很明确了，你就得到正确答案加到一起即可。

6.8 Checking types 检查类型

What happens if we call factorial and give it 1.5 as an argument?

> 如果我们让阶乘使用1.5做参数会咋样？

```
>>> factorial(1.5)
>>> factorial(1.5)
RuntimeError: Maximum recursion depth exceeded
```

It looks like an infinite recursion. How can that be? The function has a base case—when n == 0. But if n is not an integer, we can miss the base case and recurse forever.

> 看上去就像是无穷递归一样。为什么会这样？因为这个函数的基准条件是n等于0。但如果n不是一个整形变量，就会无法达到基准条件，然后无穷递归下去。

In the first recursive call, the value of n is 0.5. In the next, it is -0.5. From there, it gets smaller (more negative), but it will never be 0. We have two choices. We can try to generalize the factorial function to work with floating-point numbers, or we can make factorial check the type of its argument. The first option is called the gamma function and it's a little beyond the scope of this book. So we'll go for the second.

> 在第一次递归调用的时候，传递的n值是0.5.下一步就是-0.5.
>
> 从这开始，这个值就越来越小（就成了更小的负数了）但永远都不会到0.
>
> 我们有两种选择。我们可以尝试着把阶乘函数改写成可以用浮点数做参数的，或者也可以让阶乘函数检查一下参数类型。第一种选择就会写出一个伽玛函数，这已经超越了本书的范围。所以我们用第二种方法。

（译者注：伽玛函数（Gamma函数），也叫欧拉第二积分，是阶乘函数在实数与复数上扩展的一类函数。该函数在分析学、概率论、偏微分方程和组合数学中有重要的应用。与之有密切联系的函数是贝塔函数，也叫第一类欧拉积分。） We can use the built-in function isinstance to verify the type of the argument. While we’re at it, we can also make sure the argument is positive:

> 我们可以用内置的isinstance函数来判断参数的类型。我们也还得确定一下参数得是大于0的：

```
def factorial (n):
    if not isinstance(n, int):
        print('Factorial is only defined for integers.')
        return None
    elif n < 0:
        print('Factorial is not defined for negative integers.')
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

The first base case handles nonintegers; the second handles negative integers. In both cases, the program prints an error message and returns None to indicate that something went wrong:

> 第一个基准条件用来处理非整数；第二个用来处理负整数。在小数或者负数做参数的时候，函数就会输出错误信息，返回空到调用出来表明出错了：

```
>>> factorial('fred')
>>> factorial('fred')
Factorial is only defined for integers. None
>>> factorial(-2)
>>> factorial(-2)
Factorial is not defined for negative integers. None
```

If we get past both checks, we know that n is positive or zero, so we can prove that the recursion terminates.

> 如果两个检查都通过了，就能确定n是正整数或者0，就可以保证递归能够正确进行和终止了。

This program demonstrates a pattern sometimes called a guardian. The first two conditionals act as guardians, protecting the code that follows from values that might cause an error. The guardians make it possible to prove the correctness of the code.

> 这个程序展示了一种叫做『守卫』的模式。前两个条件就扮演了守卫的角色，避免了那些引起错误的变量。这些守卫能够保证我们的代码运行正确。

In Section 11.4 we will see a more flexible alternative to printing an error message: raising an exception.

> 在11.4我们还会看到更多的灵活的处理方式，会输出错误信息，并上报异常。

6.9 Debugging 调试

Breaking a large program into smaller functions creates natural checkpoints for debugging. If a function is not working, there are three possibilities to consider:

> 把大的程序切分成小块的函数，就自然为我们调试建立了一个个的检查点。在不工作的函数里面，有几种导致错误的可能：

- There is something wrong with the arguments the function is getting; a precondition is violated.

> 函数接收的参数可能有错，前置条件没有满足。

- There is something wrong with the function; a postcondition is violated.

> 函数本身可能有错，后置条件没有满足。

• There is something wrong with the return value or the way it is being used.

> 返回值或者返回值使用的方法可能有错。

To rule out the first possibility, you can add a print statement at the beginning of the function and display the values of the parameters (and maybe their types). Or you can write code that checks the preconditions explicitly. If the parameters look good, add a print statement before each return statement and display the return value. If possible, check the result by hand. Consider calling the function with values that make it easy to check the result (as in Section 6.2).

> 要去除第一种情况，你要在函数开头的时候添加一个print语句，来输出一下参数的值（最好加上类型）。或者你可以写一份代码来明确检查一下前置条件是否满足。
>
> 如果形式参数看上去没问题，在每一个返回语句之前都加上print语句，显示一下要返回的值。如果可以的话，尽量亲自去检查一下这些结果，自己算一算。尽量调用有值的函数，这样检查结果更方便些（比如在6.2中那样。）

If the function seems to be working, look at the function call to make sure the return value is being used correctly (or used at all!). Adding print statements at the beginning and end of a function can help make the flow of execution more visible. For example, here is a version of factorial with print statements:

> 如果函数看着没啥问题，就检查一下函数的调用，来检查一下返回值是不是有用到，确保返回值被正确使用。
>
> 在函数的开头结尾添加输出语句，能够确保整个执行流程更加可视化。比如下面就是一个有输出版本的阶乘函数：

```
def factorial(n):
    space = ' ' * (4 * n)
    print(space, 'factorial', n)
    if n == 0:
        print(space, 'returning 1')
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        print(space, 'returning', result)
        return result
```

space is a string of space characters that controls the indentation of the output. Here is the result of factorial(4) :

> space在这里是一串空格的字符串，是用来缩进输出的。下面就是4的阶乘得到的结果：

```
                             factorial 4
                     factorial 3
                 factorial 2
                 factorial 1
         factorial 0
         returning 1
            returning 1
                returning 2
                     returning 6
                        returning 24
```

If you are confused about the flow of execution, this kind of output can be helpful. It takes some time to develop effective scaffolding, but a little bit of scaffolding can save a lot of debugging.

> 如果你对执行流程比较困惑，这种输出会有一定帮助。有效率地进行脚手架开发是需要时间的，但稍微利用一下这种思路，反而能够节省调试用的时间。

6.10 Glossary 术语列表

temporary variable: A variable used to store an intermediate value in a complex calculation.

> 临时变量：用来在复杂运算过程中存储一些中间值的变量。

dead code: Part of a program that can never run, often because it appears after a return statement.

> 无效代码：一部分不会被运行的代码，一般是书现在了返回语句之后。

incremental development: A program development plan intended to avoid debugging by adding and testing only a small amount of code at a time.

> 渐进式开发：程序开发的一种方式，每次对已有的能工作的代码进行小规模的增补修改来降低调试的精力消耗。

scaffolding: Code that is used during program development but is not part of the final version.

> 脚手架代码：在程序开发期间使用的代码，但最终版本并不会包含这些代码。

guardian: A programming pattern that uses a conditional statement to check for and handle circumstances that might cause an error.

> 守卫：一种编程模式。使用一些条件语句来检验和处理一些有可能导致错误的情况。

6.11 Exercises 练习

Exercise 1 练习1

Draw a stack diagram for the following program. What does the program print?

> 为下面的程序画栈图。程序输出会是什么样的？

```
def b(z):
    prod = a(z, z)
    print(z, prod)
    return prod
def a(x, y):
    x = x + 1
    return x * y
def c(x, y, z):
    total = x + y + z
    square = b(total)**2
    return square
x = 1
y = x + 1
print(c(x, y+3, x+y))
```

Exercise 2 练习2

The Ackermann function, A(m, n), is defined:

> 阿克曼函数的定义如下：

```
A(m, n) =   n+1    if  m = 0
            A(m−1, 1)    if  m > 0  and  n = 0
            A(m−1, A(m, n−1))    if  m > 0  and  n > 0.
```

See Here. Write a function named ack that evaluates the Ackermann function. Use your function to evaluate ack(3, 4), which should be 125. What happens for larger values of m and n? Solution.

> 看一下这个连接。写一个叫做ack的函数，实现上面这个阿克曼函数。用你写出的函数来计算ack(3, 4)，结果应该是125.看看m和n更大一些会怎么样。样例代码.

Exercise 3 练习3

A palindrome is a word that is spelled the same backward and forward, like “noon” and “redivider”. Recursively, a word is a palindrome if the first and last letters are the same and the middle is a palindrome.

> 回文的词特点是正序和倒序拼写相同给，比如noon以及redivider。用递归的思路来看，回文词的收尾相同，中间部分是回文词。

The following are functions that take a string argument and return the first, last, and middle letters:

> 下面的函数是把字符串作为实际参数，然后返回函数的头部、尾部以及中间字母：

```
def first(word):
    return word[0]
def last(word):
    return word[-1]
def middle(word):
    return word[1:-1]
```

We’ll see how they work in Chapter 8.

> 第八章我们再看看他们是到底怎么工作的。

1. Type these functions into a file named palindrome.py and test them out. What happens if you call middle with a string with two letters? One letter? What about the empty string, which is written '' and contains no letters?

> 把这些函数输入到一个名字叫做palindrome.py的文件中，测试一下输出。
>
> 如果中间你使用一个只有两个字符的字符串会怎么样？一个字符的怎么样？
>
> 空字符串，比如『』没有任何字母的，怎么样？

1. Write a function called is_palindrome that takes a string argument and returns True if it is a palindrome and False otherwise. Remember that you can use the built-in function

len to check the length of a string. Solution: http://thinkpython2.com/code/palindrome_soln.py.

> 写一个名叫is_palindrome的函数，使用字符串作为实际参数，根据字符串是否为回文词来返回真假。机主，你可以用内置的len函数来检查字符串的长度。

Exercise 4 练习4

A number, a, is a power of b if it is divisible by b and a/b is a power of b. Write a function called is_power that takes parameters a and b and returns True if a is a power of b. Note: you will have to think about the base case.

> 一个数字a为b的权（power），如果a能够被b整除，并且a/b是b的权。写一个叫做 is_power 的函数接收a和b作为形式参数，如果a是b的权就返回真。注意：要考虑好基准条件。

Exercise 5 练习5

The greatest common divisor (GCD) of a and b is the largest number that divides both of them with no remainder.

> a和b的最大公约数是指能同时将这两个数整除而没有余数的数当中的最大值。

One way to find the GCD of two numbers is based on the observation that if r is the remainder when a is divided by b, then gcd(a, b) = gcd(b, r). As a base case, we can use gcd(a, 0) = a.

> 找最大公约数的一种方法是观察，如果当r是a除以b的余数，那么a和b的最大公约数与b和r的最大公约数相等。基准条件是a和0的最大公约数为a。

Write a function called gcd that takes parameters a and b and returns their greatest common divisor.

> 写一个有名叫gcd的函数，用a和b两个形式参数，返回他们的最大公约数。

Credit: This exercise is based on an example from Abelson and Sussman's Structure and Interpretation of Computer Programs.

> 致谢：这个练习借鉴了Abelson和Sussman的 计算机程序结构和解译 一书。

Chapter 7 Iteration 迭代

This chapter is about iteration, which is the ability to run a block of statements repeatedly. We saw a kind of iteration, using recursion, in Section 5.8. We saw another kind, using a for loop, in Section 4.2. In this chapter we'll see yet another kind, using a while statement. But first I want to say a little more about variable assignment.

> 这一章我们讲迭代，简单说就是指重复去运行一部分代码。在5.8的时候我们接触了一种迭代——递归。在4.2我们还学了另外一种迭代——for循环。在本章，我们会见到新的迭代方式：whie语句。但我要先再稍微讲一下变量赋值。

7.1 Reassignment 再赋值

As you may have discovered, it is legal to make more than one assignment to the same variable. A new assignment makes an existing variable refer to a new value (and stop referring to the old value).

> 你可能已经发现了，对同一个变量可以多次进行赋值。一次新的赋值使得已有的变量获得新的值（也就不再有旧的值了。）
>
> （译者注：这个其实中文很好理解，英文当中词汇逻辑关系比较紧密，但灵活程度不如中文高啊。）

```
>>> x = 5
>>> x = 5
>>> x
>>> x
5
>>> x = 7
>>> x = 7
>>> x
>>> x
7
```

The first time we display x, its value is 5; the second time, its value is 7.

> 第一次显示x的值，是5，第二次，就是7了。

Figure 7.1 shows what reassignment looks like in a state diagram.

> 图7.1表示了再赋值的操作在状态图中的样子。

At this point I want to address a common source of confusion. Because Python uses the equal sign (=) for assignment, it is tempting to interpret a statement like a = b as a mathematical proposition of equality; that is, the claim that a and b are equal. But this

interpretation is wrong.

> 这里我就要强调一下大家常发生的误解。因为Python使用单等号（=）来赋值，所以有的人会以为像a=b这样的语句就如同数学上的表达一样来表达两者相等，这种想法是错误的！

First, equality is a symmetric relationship and assignment is not. For example, in mathematics, if a=7 then 7=a. But in Python, the statement a = 7 is legal and 7 = a is not.

> 首先，数学上的等号所表示的相等是一个对称的关系，而Python中等号的赋值操作是不对称的。比如，在数学上，如果a=7，那么7=a。而在Python，a=7这个是合乎语法的，而7=a是错误的。
>
> （译者注：这里就是说Python中等号是一种单向的运算，就是把等号右边的值赋给等号左边的变量，而Python中其实也有数学上相等判断的表达式，就是双等号（==），这个是有对称性的，就是说a==b，那么b==a，或者a==3，3==a也可以。）

Also, in mathematics, a proposition of equality is either true or false for all time. If a=b now, then a will always equal b. In Python, an assignment statement can make two variables equal, but they don’t have to stay that way:

> 另外在数学上，一个相等关系要么是真，要么是假。比如a=b，那么a就会永远等于b。在Python里面，赋值语句可以让两个变量相等，但可以不始终都相等，如下所示：

```
>>> a = 5
>>> a = 5
>>> b = a     # a and b are now equal a和b相等了
>>> b = a     # a and b are now equal a和b相等了
>>> a = 3     # a and b are no longer equal 现在a和b就不相等了
>>> a = 3     # a and b are no longer equal 现在a和b就不相等了
>>> b
>>> b
5
```

The third line changes the value of a but does not change the value of b, so they are no longer equal.

> 第三行改变了a的值，但没有改变b的值，所以它们就不再相等了。

Reassigning variables is often useful, but you should use it with caution. If the values of variables change frequently, it can make the code difficult to read and debug.

> 对变量进行再赋值总是很有用的，但你用的时候要做好备注和提示。如果变量的值频繁变化，就可能让代码难以阅读和调试。

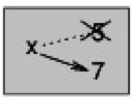


Figure 7.1: State diagram.

7.2 Updating variables 更新变量

A common kind of reassignment is an update, where the new value of the variable depends on the old.

> 最常见的一种再赋值就是对变量进行更新，这种情况下新的值是在旧值基础上进行修改得到的。

```
>>> x = x + 1
>>> x = x + 1
```

This means “get the current value of x, add one, and then update x with the new value.” If you try to update a variable that doesn’t exist, you get an error, because Python evaluates the right side before it assigns a value to x:

> 上面的语句的意思是：获取x当前的值，在此基础上加1，然后把结果作为新值赋给x。如果你对不存在的变量进行更新，你就会得到错误了，因为Python要先进行等号右边的运算，然后才能赋值给x。

```
>>> x = x + 1
>>> x = x + 1
NameError: name 'x' is not defined
```

Before you can update a variable, you have to initialize it, usually with a simple assignment:

> 在你更新一个变量之前，你要先初始化一下，一般就是简单赋值一下就可以了：

```
>>> x = 0
>>> x = 0
>>> x = x + 1
>>> x = x + 1
```

Updating a variable by adding 1 is called an increment; subtracting 1 is called a decrement.

> 对一个变量每次加1也可以叫做一种递增，每次减去1就可以叫做递减了。

7.3 The while statement 循环：While语句

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly. In a computer program, repetition is also called iteration.

> 计算机经常被用来自动执行一些重复的任务。重复同样的或者相似的任务，而不出错，这是计算机特别擅长的事情，咱们人就做不到了。在一个计算机程序里面，重复操作也被叫做迭代。

We have already seen two functions, countdown and print_n, that iterate using recursion. Because iteration is so common, Python provides language features to make it easier. One is the for statement we saw in Section 4.2. We'll get back to that later. Another is the while statement. Here is a version of countdown that uses a while statement:

> 我们已经见过两种使用了递归来进行迭代的函数：倒计时函数countdown，以及n次输出函数print_n。Python还提供了一些功能来简化迭代，因为迭代用的很普遍。其中一个就是我们在4.2中见到的for循环语句。往后我们还要复习一下它。另外一个就是while循环语句。下面就是一个使用了while循环语句来实现的倒计时函数countdown：

```
def countdown(n):
    while n > 0:
        print(n)
        n = n - 1
    print('Blastoff!')
```

You can almost read the while statement as if it were English. It means, "While n is greater than 0, display the value of n and then decrement n. When you get to 0, display the word Blastoff!"

> while循环语句读起来很容易，几乎就像是英语一样简单。这个函数的意思是：当n大于0，就输出n的值，然后对n减1，到n等于0的时候，就输出单词『Blastoff』。

More formally, here is the flow of execution for a while statement:

> 更正式一点，下面是一个while循环语句的执行流程：

1. Determine whether the condition is true or false.

> 判断循环条件的真假。

1. If false, exit the while statement and continue execution at the next statement.

> 如果是假的，退出while语句，继续运行后面的语句。

1. If the condition is true, run the body and then go back to step 1.

> 如果条件为真，执行循环体，然后再调回到第一步。

This type of flow is called a loop because the third step loops back around to the top. The body of the loop should change the value of one or more variables so that the condition becomes false eventually and the loop terminates.

> 这种类型的运行流程叫做循环，因为第三步会循环到第一步。循环体内要改变一个或者更多变量的值，这样循环条件最终才能变成假，然后循环才能终止。

Otherwise the loop will repeat forever, which is called an infinite loop. An endless source of amusement for computer scientists is the observation that the directions on shampoo, “Lather, rinse, repeat”, are an infinite loop.

> 否则的话，条件不能为假，循环不能停止，这就叫做无限循环。计算机科学家有一个笑话，就是看到洗发液的说明：起泡，冲洗，重复；这就是一个无限循环。

In the case of countdown, we can prove that the loop terminates: if n is zero or negative, the loop never runs. Otherwise, n gets smaller each time through the loop, so eventually we have to get to 0.

> 在倒计时函数countdown里面，咱们能够保证有办法让循环终止：只要n小于等于0了，循环就不进行了。否则的话，n每次就会通过循环来递减，最终还是能到0的。

For some other loops, it is not so easy to tell. For example:

> 其他一些循环就不那么好描述了。比如：

```
def sequence(n):
    while n != 1:
        print(n)
        if n % 2 == 0:            # n is even
            n = n / 2
        else:                     # n is odd
            n = n*3 + 1
```

The condition for this loop is n != 1, so the loop will continue until n is 1, which makes the condition false.

> 这个循环的判断条件是n不等于1，所以循环一直进行，直到n等于1了，条件为假，就不再循环了。

Each time through the loop, the program outputs the value of n and then checks whether it is even or odd. If it is even, n is divided by 2. If it is odd, the value of n is replaced with n*3 + 1. For example, if the argument passed to sequence is 3, the resulting values of n are 3, 10, 5, 16, 8, 4, 2, 1.

> 每次循环的时候，程序都输出n的值，然后检查一下是偶数还是奇数。如果是偶数，就把n除以2。如果是奇数，就把n替换为n乘以3再加1的值。比如让这个函数用3做参数，也就是sequence(3)，得到的n的值就依次为：3, 10, 5, 16, 8, 4, 2, 1.

Since n sometimes increases and sometimes decreases, there is no obvious proof that n will ever reach 1, or that the program terminates. For some particular values of n, we can prove termination. For example, if the starting value is a power of two, n will be even every time

through the loop until it reaches 1. The previous example ends with such a sequence, starting with 16.

> 有时候n在增加，有时候n在降低，所以没有明显证据表明n最终会到1而程序停止。对于一些特定的n值，我们能够确保循环的终止。例如如果起始值是一个2的倍数，n每次循环过后依然是偶数，直到到达1位置。之前的例子都最终得到了一个数列，从16开始的就是了。

The hard question is whether we can prove that this program terminates for all positive values of n. So far, no one has been able to prove it or disprove it! See WikiPedia

> 真正的难题是，我们能否证明这个程序对任意正数的n值都能终止循环。目前为止，没有人能够证明或者否定这个命题。

参考维基百科 As an exercise, rewrite the function print_n from Section 5.8 using iteration instead of recursion.

> 做一个练习，把5.8里面的那个n次打印函数print_n用迭代的方法来实现。

7.4 break 中断

Sometimes you don’t know it’s time to end a loop until you get half way through the body. In that case you can use the break statement to jump out of the loop.

> 有时候你不知道啥时候终止循环，可能正好在中间循环体的时候要终止了。这时候你就可以用break语句来跳出循环。

For example, suppose you want to take input from the user until they type done. You could write:

> 比如，假设你要让用户输入一些内容，当他们输入done的时候结束。你就可以用如下的方法实现：

```
while True:
    line = input('> ')
    if line == 'done':
        break
        print(line)
    print('Done!')
```

The loop condition is True, which is always true, so the loop runs until it hits the break statement.

> 循环条件就是true，意味条件总是真的，所以循环就一直进行，一直到触发了break语句才跳出。

Each time through, it prompts the user with an angle bracket. If the user types done, the break statement exits the loop. Otherwise the program echoes whatever the user types and goes back to the top of the loop. Here's a sample run:

> 每次循环的时候，程序都会有一个大于号>来提示用户。如果用输入了done，break语句就会让程序跳出循环。否则的话程序会重复用户输入的内容，然后回到循环的头部。下面就是一个简单的运行例子：

```
>>>not done
>>>not done
>>>not done
>>>not done
>>>done
>>>done
Done!
```

This way of writing while loops is common because you can check the condition anywhere in the loop (not just at the top) and you can express the stop condition affirmatively ("stop when this happens") rather than negatively ("keep going until that happens").

> 这种while循环的写法很常见，因为这样你可以在循环的任何一个部位对条件进行检测，而不仅仅是循环的头部，你可以确定地表达循环停止的条件（在这种情况下就停止了），而不是消极地暗示『程序会一直运行，直到某种情况』。

7.5 Square roots 平方根

Loops are often used in programs that compute numerical results by starting with an approximate answer and iteratively improving it.

> 循环经常被用于进行数值运算的程序中，这种程序往往是有一个近似值作为初始值，然后逐渐迭代去改进以接近真实值。

For example, one way of computing square roots is Newton's method. Suppose that you want to know the square root of a. If you start with almost any estimate, x, you can compute a better estimate with the following formula:

> 比如，可以用牛顿法来计算平方根。加入你要知道一个数a的平方根。如果你用任意一个估计值x来开始，你可以用下面的公式获得一个更接近的值：

```
y = \frac{x + \frac{a}{x}}{2}
```

For example, if a is 4 and x is 3:

> 比如，如果a是3，x设为3：

```
>>> a = 4
>>> a = 4
>>> x = 3
>>> x = 3
>>> y = (x + a/x) / 2
>>> y = (x + a/x) / 2
>>> y
>>> y
2.16666666667
```

The result is closer to the correct answer (square root of 4 is 2). If we repeat the process with the new estimate, it gets even closer:

> 得到的结果比初始值3更接近真实值（4的平方根是2）。如果我们用这个结果做新的估计值来重复这个操作，结果就更加接近了：

```
>>> x = y
>>> x = y
>>> y = (x + a/x) / 2
>>> y = (x + a/x) / 2
>>> y 2.00641025641
>>> y 2.00641025641
```

After a few more updates, the estimate is almost exact:

> 这样进行一些次重复之后，估计值就几乎很准确了：

```
>>> x = y
>>> x = y
>>> y = (x + a/x) / 2
>>> y = (x + a/x) / 2
>>> y 2.00001024003
>>> y 2.00001024003
>>> x = y
>>> x = y
>>> y = (x + a/x) / 2
>>> y = (x + a/x) / 2
>>> y 2.00000000003
>>> y 2.00000000003
```

In general we don't know ahead of time how many steps it takes to get to the right answer, but we know when we get there because the estimate stops changing:

> 一般情况下，我们不能提前知道到达正确结果需要多长时间，但是当估计值不再有明显变化的时候我们就知道了：

```
>>> x = y
>>> x = y
>>> y = (x + a/x) / 2
>>> y = (x + a/x) / 2
>>> y 2.0
>>> y 2.0
>>> x = y
>>> x = y
>>> y = (x + a/x) / 2
>>> y = (x + a/x) / 2
>>> y 2.0
>>> y 2.0
```

When y == x, we can stop. Here is a loop that starts with an initial estimate, x, and improves it until it stops changing:

> 当y和x相等的时候，我们就可以停止了。下面这个循环中，用一个初始值x来开始循环，然后进行改进，一直到x的值不再变化为止：

```
while True:
    print(x)
    y = (x + a/x) / 2
    if y == x:
        break
    x = y
```

For most values of a this works fine, but in general it is dangerous to test float equality. Floating-point values are only approximately right: most rational numbers, like 1/3, and irrational numbers, like √2, can't be represented exactly with a float.

> 对大多数值来说，这个循环都挺管用的，但一般来说用浮点数来测试等式是很危险的。浮点数的值只能是近似正确：大多数的有理数，比如1/3，以及无理数，比如根号2，都不能用浮点数来准确表达的。

Rather than checking whether x and y are exactly equal, it is safer to use the built-in function abs to compute the absolute value, or magnitude, of the difference between them:

> 相比之下，与其对比x和y是否精确相等，倒不如以下方法更安全：用内置的绝对值函数来计算一下差值的绝对值，也叫做数量级。

```
if abs(y-x) < epsilon:
    break
```

Where epsilon has a value like 0.0000001 that determines how close is close enough.

> 这里可以让epsilon的值为like 0.0000001，差值比这个小就说明已经足够接近了。

7.6 Algorithms 算法

Newton’s method is an example of an algorithm: it is a mechanical process for solving a category of problems (in this case, computing square roots).

> 牛顿法是算法的一个例子：通过一系列机械的步骤来解决一类问题（在本章中是用来计算平方根）。

To understand what an algorithm is, it might help to start with something that is not an algorithm. When you learned to multiply single-digit numbers, you probably memorized the multiplication table. In effect, you memorized 100 specific solutions. That kind of knowledge is not algorithmic.

> 要理解算法是什么，先从一些不是算法的内容来开始也许会有所帮助。当你学个位数字乘法的时候，你可能要背下来整个乘法表。实际上你记住了100个特定的算式。这种知识就不是算法。

But if you were “lazy”, you might have learned a few tricks. For example, to find the product of n and 9, you can write n−1 as the first digit and 10−n as the second digit. This trick is a general solution for multiplying any single-digit number by 9. That’s an algorithm!

> 但如果你很『懒』，你就可能会有一些小技巧。比如找到一个n与9的成绩，你可以把n-1写成第一位，10-n携程第二位。这个技巧是应对任何个位数字乘以9的算式。这就是一个算法了！

Similarly, the techniques you learned for addition with carrying, subtraction with borrowing, and long division are all algorithms. One of the characteristics of algorithms is that they do not require any intelligence to carry out. They are mechanical processes where each step follows from the last according to a simple set of rules.

> 相似地，你学过的进位的加法，借位的减法，以及长除法，都是算法。这些算法的一个共同特点就是不需要任何智力就能进行。它们都是机械的过程，每一步都跟随上一步，遵循着很简单的一套规则。

Executing algorithms is boring, but designing them is interesting, intellectually challenging, and a central part of computer science.

> 执行算法是很无聊的，但设计算法很有趣，是智力上的一种挑战，也是计算机科学的核心部分。

Some of the things that people do naturally, without difficulty or conscious thought, are the hardest to express algorithmically. Understanding natural language is a good example. We all do it, but so far no one has been able to explain how we do it, at least not in the form of an algorithm.

> 有的事情人们平时做起来很简单，甚至都不用思考，这些事情往往最难用算法来表达。比如理解自然语言就是个例子。我们都能理解自然语言，但目前为止还没有人能解释我们到底是怎么做到的，至少没有人把这个过程归纳出算法的形式。

7.7 Debugging 调试

As you start writing bigger programs, you might find yourself spending more time debugging. More code means more chances to make an error and more places for bugs to hide.

> 现在你已经开始写一些比较大的程序了，你可能发现自己比原来花更多时间来调试了。代码越多，也就意味着出错的可能也越大了，bug也有了更多的藏身之处了。

One way to cut your debugging time is “debugging by bisection”. For example, if there are 100 lines in your program and you check them one at a time, it would take 100 steps. Instead, try to break the problem in half. Look at the middle of the program, or near it, for an intermediate value you can check. Add a print statement (or something else that has a verifiable effect) and run the program.

> 『对折调试』是一种节省调试时间的方法。比如，如果你的程序有100行，你检查一遍就要大概100步了。而对折方法就是把程序分成两半。看程序中间位置，或者靠近中间位置的，检查一些中间值。在这些位置添加一些print语句（或者其他能够能起到验证效果的东西），然后运行程序。

If the mid-point check is incorrect, there must be a problem in the first half of the program. If it is correct, the problem is in the second half.

> 如果中间点检查出错了，那必然是程序的前半部分有问题。如果中间点没调试，那问题就是在后半段了。

Every time you perform a check like this, you halve the number of lines you have to search. After six steps (which is fewer than 100), you would be down to one or two lines of code, at least in theory.

> 每次你都这样检查，你就让你要检查的代码数量减半了。一般六步之后（远小于100次了），理论上你就差不多已经到代码的末尾一两行了。

In practice it is not always clear what the “middle of the program” is and not always possible to check it. It doesn’t make sense to count lines and find the exact midpoint. Instead, think about places in the program where there might be errors and places where it is easy to put a check. Then choose a spot where you think the chances are about the same that the bug is before or after the check.

> 在实际操作当中，程序中间位置并不是总那么明确，也未必就很容易去检查。所以不用数行数来找确定的中间点。相反的，只要考虑一下程序中哪些地方容易调试，然后哪些地方进行检验比较容易就行了。然后你就在你考虑好的位置检验一下看看bug是在那个位置之前还是之后。

7.8 Glossary 术语列表

reassignment: Assigning a new value to a variable that already exists.

> 再赋值：对一个已经存在的有值变量赋予一个新的值。

update: An assignment where the new value of the variable depends on the old.

> 更新：根据一个变量的旧值，进行一定的修改，再赋值给这个变量。

initialization: An assignment that gives an initial value to a variable that will be updated.

> 初始化：给一个变量初始值，以便于后续进行更新。

increment: An update that increases the value of a variable (often by one).

> 递增：每次给一个变量增加一定的值（一般是加1）

decrement: An update that decreases the value of a variable.

> 递减：每次给一个变量减去一定的值。

iteration: Repeated execution of a set of statements using either a recursive function call or a loop.

> 迭代：重复执行一系列语句，使用递归函数调用的方式，或者循环的方式。

infinite loop: A loop in which the terminating condition is never satisfied.

> 无限循环：终止条件永远无法满足的循环。

algorithm: A general process for solving a category of problems.

> 算法：解决某一类问题的一系列通用的步骤。

7.9 Exercises 练习

Exercise 1 练习1

Copy the loop from Section 7.5 and encapsulate it in a function called mysqrt that takes a as a parameter, chooses a reasonable value of x, and returns an estimate of the square root of a.

> 从7.5复制一个循环，然后改写成名字叫做mysqrt的函数，该函数用一个a作为参数，选择一个适当的起始值x，然后返回a的平方根的近似值。

To test it, write a function named test_square_root that prints a table like this:

> 测试这个函数，写一个叫做test_suqare_root的函数来输出以下这样的表格：

a	mysqrt(a)	math.sqrt(a)	diff
1.0	1.0	1.0	0.0
2.0	1.41421356237	1.41421356237	2.22044604925e-16
3.0	1.73205080757	1.73205080757	0.0
4.0	2.0	2.0	0.0
5.0	2.2360679775	2.2360679775	0.0
6.0	2.44948974278	2.44948974278	0.0
7.0	2.64575131106	2.64575131106	0.0
8.0	2.82842712475	2.82842712475	4.4408920985e-16
9.0	3.0	3.0	0.0

The first column is a number, a; the second column is the square root of acomputed with mysqrt; the third column is the square root computed by math.sqrt; the fourth column is the absolute value of the difference between the two estimates.

> 第一列是数a；第二列是咱们自己写的函数mysqrt计算出来的平方根，第三行是用Python内置的math.sqrt函数计算的平方根，最后一行是这两者的差值的绝对值。

Exercise 2 练习2

The built-in function eval takes a string and evaluates it using the Python interpreter. For example:

> Python的内置函数eval接收字符串作为参数，然后用Python的解释器来运行。例如：

```
>>> eval('1 + 2 * 3')
>>> eval('1 + 2 * 3')
7
>>> import math
>>> import math
>>> eval('math.sqrt(5)')
>>> eval('math.sqrt(5)')
2.2360679774997898
>>> eval('type(math.pi)')
>>> eval('type(math.pi)')
<class 'float'>
```

Write a function called eval_loop that iteratively prompts the user, takes the resulting input and evaluates it using eval, and prints the result.

> 写一个叫做eval_loop的函数，交互地提醒用户，获取输入，然后用eval对输入进行运算，把结果打印出来。

It should continue until the user enters 'done', and then return the value of the last expression it evaluated.

> 这个程序要一直运行，直到用户输入『done』才停止，然后输出最后一次计算的表达式的值。

Exercise 3 练习3

The mathematician Srinivasa Ramanujan found an infinite series that can be used to generate a numerical approximation of 1 / π:

> 传奇的数学家拉马努金发现了一个无穷级数（1914年的论文），能够用来计算圆周率倒数的近似值：

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{99^2} \sum_{k=0}^{\infty} \frac{(4k)!(26390k + 1103)}{(k!)^4 396^{4k}}$$

（译者注：这位拉马努金是一位非常杰出的数学家，自学成才，以数论为主要研究内容，可惜33岁的时候就英年早逝。他被哈代誉为超越希尔伯特的天才。） Write a function called estimate_pi that uses this formula to compute and return an estimate of π. It should use a while loop to compute terms of the summation until the last term is smaller than 1e-15 (which is Python notation for 10^{-15}). You can check the result by comparing it to math.pi. Solution

> 写一个名叫estimate_pi的函数，用上面这个方程来计算并返回一个圆周率π的近似值。要使用一个while循环来计算出总和的每一位，最后一位要小于10的-15次方。你可以对比一下计算结果和Python内置的math.pi。
>
> 样例代码

Chapter 8 Strings 字符串

Strings are not like integers, floats, and booleans. A string is a sequence, which means it is an ordered collection of other values. In this chapter you'll see how to access the characters that make up a string, and you'll learn about some of the methods strings provide.

> 字符串和整形、浮点数以及布尔值很不一样。一个字符串是一个序列，意味着是对其他值的有序排列。在本章你将学到如何读取字符串中的字符，你还会学到一些字符串相关的方法。

8.1 A string is a sequence 字符串是序列

A string is a sequence of characters. You can access the characters one at a time with the bracket operator:

> 字符串就是一串有序的字符。你可以通过方括号操作符，每次去访问字符串中的一个字符：

```
>>> fruit = 'banana'
>>> fruit = 'banana'
>>> letter = fruit[1]
>>> letter = fruit[1]
```

The second statement selects character number 1 from fruit and assigns it to letter.

> 第二个语句选择了 fruit 这个字符串的序号为1的字符，并把这个字符赋值给了 letter 这个变量。

（译者注：思考一下这里的 letter 是一个什么类型的变量。） The expression in brackets is called an index. The index indicates which character in the sequence you want (hence the name). But you might not get what you expect:

> 方括号内的内容叫做索引。索引指示了你所指定的字符串中字符的位置（就跟名字差不多）。
>
> 但你可能发现得到的结果和你预期的有点不一样：

```
>>> letter
>>> letter
'a'
```

For most people, the first letter of 'banana' is b, not a. But for computer scientists, the index is an offset from the beginning of the string, and the offset of the first letter is zero.

> 大多数人都认为banana 的第『1』个字符应该是 b，而不是 a。但对于计算机科学家来说，索引是字符串从头的偏移量，所以真正的首字母偏移量应该是0.

```
>>> letter = fruit[0]
>>> letter = fruit[0]
>>> letter
>>> letter
'b'
```

So b is the 0th letter (“zero-eth”) of 'banana', a is the 1th letter (“one-eth”), and n is the 2th letter (“two-eth”).

> 所以 b 就是字符串 banana 的第『0』个字符，而 a 是第『1』个，n 就是第『2』个了。

As an index you can use an expression that contains variables and operators:

> 你可以在方括号内的索引中使用变量和表达式：

```
>>> i = 1
>>> i = 1
>>> fruit[i]
>>> fruit[i]
'a'
>>> fruit[i+1]
>>> fruit[i+1]
'n'
```

But the value of the index has to be an integer. Otherwise you get:

> 但要注意的事，索引的值必须是整形的。否则你就会遇到类型错误了：

```
>>> letter = fruit[1.5]
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

8.2 len 长度

len is a built-in function that returns the number of characters in a string:

> len 是一个内置函数，会返回一个字符串中字符的长度：

```
>>> fruit = 'banana'
>>> fruit = 'banana'
>>> len(fruit) 6
>>> len(fruit) 6
```

To get the last letter of a string, you might be tempted to try something like this:

> 要得到一个字符串的最后一个字符，你可能会想到去利用 len 函数：

```
>>> length = len(fruit)
>>> length = len(fruit)
>>> last = fruit[length]
>>> last = fruit[length]
IndexError: string index out of range
```

The reason for the IndexError is that there is no letter in 'banana' with the index 6. Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, you have to subtract 1 from length:

> 出现索引错误的原因就是banana 这个字符串在第『6』个位置是没有字母的。因为我们从0开始数，所以这一共6个字母的顺序是0到5号。因此要得到最后一次字符，你需要在字符串长度的基础上减去1才行：

```
>>> last = fruit[length-1]
>>> last = fruit[length-1]
>>> last
>>> last
'a'
```

Or you can use negative indices, which count backward from the end of the string. The expression fruit[-1] yields the last letter, fruit[-2] yields the second to last, and so on.

> 或者你也可以用负数索引，意思就是从字符串的末尾向前数几位。fruit[-1]这个表达式给你最后一个字符，fruit[-2]给出倒数第二个，依此类推。

8.3 Traversal with a for loop 用 for 循环进行遍历

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a traversal. One way to write a traversal is with a while loop:

> 很多计算过程都需要每次从一个字符串中拿一个字符。一般都是从头开始，依次得到每个字符，然后做点处理，然后一直到末尾。这种处理模式叫遍历。写一个遍历可以使用 while 循环：

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

This loop traverses the string and displays each letter on a line by itself. The loop condition is index < len(fruit), so when index is equal to the length of the string, the condition is false, and the body of the loop doesn't run. The last character accessed is the one with the index

len(fruit)-1, which is the last character in the string.

> 这个循环遍历了整个字符串，然后它再把买一个字符显示在一行上面。循环条件是 index 这个变量小于字符串 fruit 的长度，所以当 index 与字符串长度相等的时候，条件就不成立了，循环体就不运行了。最后一个字符被获取的时候，index 正好是len(fruit)-1，这就已经是该字符串的最后一个字符了。

As an exercise, write a function that takes a string as an argument and displays the letters backward, one per line.

> 下面就练习一下了，写一个函数，接收一个字符串做参数，然后倒序显示每一个字符，每行显示一个。

Another way to write a traversal is with a for loop:

> 另外一种遍历的方法就是 for 循环了：

```
for letter in fruit:
    print(letter)
```

Each time through the loop, the next character in the string is assigned to the variable letter. The loop continues until no characters are left.

> 每次循环之后，字符串中的下一个字符都会赋值给变量 letter。循环在进行到没有字符剩余的时候就停止了。

The following example shows how to use concatenation (string addition) and a for loop to generate an abecedarian series (that is, in alphabetical order).

> 下面的例子展示了如何使用级联（字符串加法）以及一个 for 循环来生成一个简单的序列（用字母表顺序）。

In Robert McCloskey's book Make Way for Ducklings, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. This loop outputs these names in order:

> 在 Robert McCloskey 的一本名叫《Make Way for Ducklings》的书中，小鸭子的名字依次为：Jack, Kack, Lack, Mack, Nack, Ouack, Pack, 和Quack。下面这个循环会依次输出他们的名字：

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'
for letter in prefixes:
    print(letter + suffix)
```

The output is:

> 输出结果如下：

```
Jack Kack Lack Mack Nack Oack Pack Qack
```

Of course, that's not quite right because "Ouack" and "Quack" are misspelled. As an exercise, modify the program to fix this error.

> 当然了，有点不准确的地方，因为有"Ouack"和 "Quack"两处拼写错了。做个练习，修改一下程序，改正这个错误。

8.4 String slices 字符串切片

A segment of a string is called a slice. Selecting a slice is similar to selecting a character:

> 字符串的一段叫做切片。从字符串中选择一部分做切片，与选择一个字符有些相似：

```
>>> s = 'Monty Python'
>>> s = 'Monty Python'
>>> s[0:5]
>>> s[0:5]
'Monty'
>>> s[6:12]
>>> s[6:12]
'Python'
```

The operator [n:m] returns the part of the string from the "n-eth" character to the "m-eth" character, including the first but excluding the last. This behavior is counter intuitive, but it might help to imagine the indices pointing between the characters, as in Figure 8.1.

> [n:m]这种操作符，会返回字符串中从第『n』个到第『m』个的字符，包含开头的第『n』个，但不包含末尾的第『m』个。这个设计可能有点违背直觉，但可能有助于想象这个切片在字符串中的方向，如图8.1。

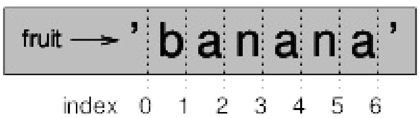


Figure 8.1: Slice indices.

If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string:

> 如果你忽略了第一个索引（就是冒号前面的那个），切片会默认从字符串头部开始。如果你忽略了第二个索引，切片会一直包含到最后一位：

```
>>> fruit = 'banana'
>>> fruit = 'banana'
>>> fruit[:3]
>>> fruit[:3]
'ban'
>>> fruit[3:]
>>> fruit[3:]
'ana'
```

If the first index is greater than or equal to the second the result is an empty string, represented by two quotation marks:

> 如果两个索引相等，得到的就是空字符串了，用两个单引号来表示：

```
>>> fruit = 'banana'
>>> fruit = 'banana'
>>> fruit[3:3]
>>> fruit[3:3]
''
```

An empty string contains no characters and has length 0, but other than that, it is the same as any other string. Continuing this example, what do you think fruit[:] means? Try it and see.

> 空字符串不包含字符，长度为0，除此之外，都与其他字符串是一样的。
>
> 那么来练习一下，你觉得 fruit[:]这个是什么意思？在程序中试试吧。

8.5 Strings are immutable 字符串不可修改

It is tempting to use the [] operator on the left side of an assignment, with the intention of changing a character in a string. For example:

> 大家总是有可能想试试把方括号在赋值表达式的等号左侧，试图去更改字符串中的某一个字符。比如：

```
>>> greeting = 'Hello, world!'
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
>>> greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
```

The “object” in this case is the string and the “item” is the character you tried to assign. For now, an object is the same thing as a value, but we will refine that definition later (Section 10.10).

> 『object』是对象的意思，这里指的是字符串类 string，然后『item』是指你试图赋值的字符串中的字符。目前来说，一个对象就跟一个值差不多，但后续在第十章第十节我们再对这个定义进行详细讨论。

The reason for the error is that strings are immutable, which means you can't change an existing string. The best you can do is create a new string that is a variation on the original:

> 产生上述错误的原因是字符串是不能被修改的，这意味着你不能对一个已经存在的字符串进行任何改动。你顶多也就能建立一个新字符串，新字符串可以基于旧字符串进行一些改动。

```
>>> greeting = 'Hello, world!'
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> new_greeting = 'J' + greeting[1:]
>>> new_greeting
>>> new_greeting
'Jello, world!'
```

This example concatenates a new first letter onto a slice of greeting. It has no effect on the original string.

> 上面的例子中，对 greeting 这个字符串进行了切片，然后添加了一个新的首字母过去。这并不会对原始字符串有任何影响。（译者注：也就是 greeting 这个字符串的值依然是原来的值，是不可改变的。）

8.6 Searching 搜索

What does the following function do?

> 下面这个函数是干啥的？

```
def find(word, letter):
    index = 0
    while index < len(word):
        if word[index] == letter:
            return index
        index = index + 1
            return -1
```

In a sense, find is the inverse of the [] operator. Instead of taking an index and extracting the corresponding character, it takes a character and finds the index where that character appears. If the character is not found, the function returns -1.

> 简单来说，find 函数，也就是查找，是方括号操作符[]的逆运算。方括号是之道索引然后提取对应的字符，而查找函数是选定一个字符去查找这个字符出现的索引位置。如果字符没有被报道，函数就返回-1。

This is the first example we have seen of a return statement inside a loop. If word[index] == letter, the function breaks out of the loop and returns immediately. If the character doesn't appear in the string, the program exits the loop normally and returns -1.

> 这是我们见过的第一个返回语句位于循环体内的例子。如果word[index]等于letter，函数就跳出循环立刻返回。如果字符在字符串里面没出现，程序正常退出循环并且返回-1。

This pattern of computation—traversing a sequence and returning when we find what we are looking for—is called a search. As an exercise, modify find so that it has a third parameter, the index in word where it should start looking.

> 这种计算-遍历一个序列然后返回我们要找的东西的模式就叫做搜索了。
>
> 做个练习，修改一下 find 函数，加入第三个参数，这个参数为查找开始的字符串位置。

8.7 Looping and counting 循环和计数

The following program counts the number of times the letter a appears in a string:

> 下面这个程序计算了字母 a 在一个字符串中出现的次数：

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
    count = count + 1
    print(count)
```

This program demonstrates another pattern of computation called a counter. The variable count is initialized to 0 and then incremented each time an a is found. When the loop exits, count contains the result—the total number of a's.

> 这一程序展示了另外一种计算模式，叫做计数。变量 count 被初始化为0，然后每次在字符串中找到一个 a，就让 count 加1.当循环退出的时候，count 就包含了 a 出现的总次数。

As an exercise, encapsulate this code in a function named count, and generalize it so that it accepts the string and the letter as arguments. Then rewrite the function so that instead of traversing the string, it uses the three-parameter version of find from the previous section.

> 做个练习，把上面的代码封装进一个名叫 count 的函数中，泛化一下，一遍让他接收任何字符串和字幕作为参数。
>
> 然后再重写一下这个函数，这次不再让它遍历整个字符串，而使用上一节中练习的三参数版本的 find 函数。

8.8 String methods 字符串方法

Strings provide methods that perform a variety of useful operations. A method is similar to a function—it takes arguments and returns a value—but the syntax is different. For example, the method upper takes a string and returns a new string with all uppercase letters. Instead of the function syntax upper(word), it uses the method syntax word.upper().

> 字符串提供了一些方法，这些方法能够进行很多有用的操作。方法和函数有些类似，也接收参数然后返回一个值，但语法稍微不同。比如，upper 这个方法就读取一个字符串，返回一个全部为大写字母的新字符串。
>
> 与函数的 upper(word)语法不同，方法的语法是 word.upper()。

```
>>> word = 'banana'
>>> word = 'banana'
>>> new_word = word.upper()
>>> new_word = word.upper()
>>> new_word 'BANANA'
>>> new_word 'BANANA'
```

This form of dot notation specifies the name of the method, upper, and the name of the string to apply the method to, word. The empty parentheses indicate that this method takes no arguments.

> 这种用点号分隔的方法表明了使用的方法名字为 upper，使用这个方法的字符串的名字为 word。后面括号里面是空白的，表示这个方法不接收参数。

A method call is called an invocation; in this case, we would say that we are invoking upper on word.

> 方法的调用被叫做——调用（译者注：这真是扯淡，中文都混淆成调用，英文里面 invocation 和 invoke 都有祈祷的意思，和 call 有显著的意义差别，但中文都混淆成调用，这种例子不胜枚举，所以大家尽量多读原版作品。）；在这里，我们就说调用了 word 的 upper 方法。

As it turns out, there is a string method named find that is remarkably similar to the function we wrote:

> 结果我们发现string 有一个方法叫做 find，跟我们写过的函数 find 有惊人的相似：

```
>>> word = 'banana'
>>> word = 'banana'
>>> index = word.find('a')
>>> index = word.find('a')
>>> index
>>> index
1
```

In this example, we invoke find on word and pass the letter we are looking for as a parameter.

> 在这里我们调用了 word 的 find 方法，然后给定了我们要找的字母 a 作为一个参数。

Actually, the find method is more general than our function; it can find substrings, not just characters:

> 实际上，这个 find 方法比我们的 find 函数功能更通用；它不仅能查找字符，还能查找字符串：

```
>>> word.find('na')
>>> word.find('na')
2
```

By default, find starts at the beginning of the string, but it can take a second argument, the index where it should start:

> 默认情况下 find 方法从字符串的开头来查找，不过可以给它一个第二个参数，让它从指定位置查找：

```
>>> word.find('na', 3)
>>> word.find('na', 3)
4
```

This is an example of an optional argument; find can also take a third argument, the index where it should stop:

> 这是一个可选参数的例子；find 方法还能接收第三个参数，可以指定查找终止的位置：

```
>>> name = 'bob'
>>> name = 'bob'
>>> name.find('b', 1, 2)
>>> name.find('b', 1, 2)
-1
```

This search fails because b does not appear in the index range from 1 to 2, not including 2. Searching up to, but not including, the second index makes find consistent with the slice operator.

> 这个搜索失败了，因为 b 并没有在索引1到2且不包括2的字符中间出现。搜索到指定的第三个变量作为索引的位置，但不包括该位置，这就让 find 方法与切片操作符相一致。

8.9 The in operator 运算符 in

The word in is a boolean operator that takes two strings and returns True if the first appears as a substring in the second:

> in 这个词在字符串操作中是一个布尔操作符，它读取两个字符串，如果前者的字符串为后者所包含，就返回真，否则为假：

```
>>> 'a' in 'banana'
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
>>> 'seed' in 'banana'
False
```

For example, the following function prints all the letters from word1 that also appear in word2:

> 举个例子，下面的函数显示所有同时在 word1和 word2当中出现的字母：

```
def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print(letter)
```

With well-chosen variable names, Python sometimes reads like English. You could read this loop, “for (each) letter in (the first) word, if (the) letter (appears) in (the second) word, print (the) letter.” Here’s what you get if you compare apples and oranges:

> 选好变量名的话，Python 有时候读起来就跟英语差不多。你读一下这个循环，就能发现，『对第一个 word 当中的每一个字母letter，如果这个字母也在第二个 word 当中出现，就输出这个字母 letter。』

```
>>> in_both('apples', 'oranges')
>>> in_both('apples', 'oranges')
a e s
```

8.10 String comparison 字符串对比

The relational operators work on strings. To see if two strings are equal:

> 关系运算符对于字符串来说也可用。比如可以看看两个字符串是不是相等：

```
if word == 'banana':
    print('All right, bananas.')
```

Other relational operations are useful for putting words in alphabetical order:

> 其他的关系运算符可以来把字符串按照字母表顺序排列：

```
if word < 'banana':
    print('Your word, ' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word, ' + word + ', comes after banana.')
else:
    print('All right, bananas.')
```

Python does not handle uppercase and lowercase letters the same way people do. All the uppercase letters come before all the lowercase letters, so: Your word, Pineapple, comes before banana.

> Python 对大小写字母的处理与人类常规思路不同。所有大写字母都在小写字母之前，所以顺序上应该是：
>
> Your word，然后是 Pineapple，然后才是 banana。

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison. Keep that in mind in case you have to defend yourself against a man armed with a Pineapple.

> 一个解决这个问题的普遍方法是把字符串转换为标准格式，比如都转成小写的，然后再进行比较。一定要记得哈，以免你遇到一个用 Pineapple 武装着自己的家伙的时候手足无措。

8.11 Debugging 调试

When you use indices to traverse the values in a sequence, it is tricky to get the beginning and end of the traversal right. Here is a function that is supposed to compare two words and return True if one of the words is the reverse of the other, but it contains two errors:

> 使用索引来遍历一个序列中的值的时候，弄清楚遍历的开头和结尾很不容易。下面这个函数用来对比两个单词，如果一个是另一个的倒序就返回真，但这个函数代码中有两处错误：

```
def is_reverse(word1, word2):
    if len(word1) != len(word2):
        return False
    i = 0
    j = len(word2)
    while j > 0:
        if word1[i] != word2[j]:
            return False
        i = i+1
        j = j-1
            return True
```

The first if statement checks whether the words are the same length. If not, we can return False immediately. Otherwise, for the rest of the function, we can assume that the words are the same length. This is an example of the guardian pattern in Section 6.8.

> 第一个 if 语句是检查两个词的长度是否一样。如果不一样长，当场就返回假。对函数其余部分，我们假设两个单词一样长。这里用到了守卫模式，在第6章第8节我们提到过。

i and j are indices: i traverses word1 forward while j traverses word2 backward. If we find two letters that don't match, we can return False immediately. If we get through the whole loop and all the letters match, we return True. If we test this function with the words “pots” and “stop”, we expect the return value True, but we get an IndexError:

> i 和 j 都是索引：i 从头到尾遍历单词 word1，而 j 逆向遍历单词word2.如果我们发现两个字母不匹配，就可以立即返回假。如果经过整个循环，所有字母都匹配，就返回真。
>
> 如果我们用这个函数来处理单词『pots』和『stop』，我们希望函数返回真，但得到的却是索引错误：

```
>>> is_reverse('pots', 'stop')
>>> is_reverse('pots', 'stop')
 ...   File "reverse.py", line 15, in is_reverse     if word1[i] != word2[j]: IndexError:
```

For debugging this kind of error, my first move is to print the values of the indices immediately before the line where the error appears.

> 为了改正这个错误，第一步就是在出错的那行之前先输出索引的值。

```
while j > 0:
    print(i, j)        # print here
if word1[i] != word2[j]:
    return False
    i = i+1
    j = j-1
```

Now when I run the program again, I get more information:

> 然后我再次运行函数，得到更多信息了：

```
>>> is_reverse('pots', 'stop')
>>> is_reverse('pots', 'stop')
0 4
... IndexError: string index out of range
```

The first time through the loop, the value of j is 4, which is out of range for the string 'pots'. The index of the last character is 3, so the initial value for j should be len(word2)-1. If I fix that error and run the program again, I get:

> 第一次循环完毕的时候，j 的值是4，这超出了『pots』这个字符串的范围了（译者注：应该是0-3）。最后一个索引应该是3，所以 j 的初始值应该是 len(word2)-1。

```
>>> is_reverse('pots', 'stop')
>>> is_reverse('pots', 'stop')
0 3 1 2 2 1
True
```

This time we get the right answer, but it looks like the loop only ran three times, which is suspicious. To get a better idea of what is happening, it is useful to draw a state diagram. During the first iteration, the frame for is_reverse is shown in Figure 8.2.

> 这次我们得到了正确的结果，但似乎循环只走了三次，这有点奇怪。为了弄明白带到怎么回事，我们可以画一个状态图。在第一次迭代的过程中，is_reverse 的框架如图8.2所示。

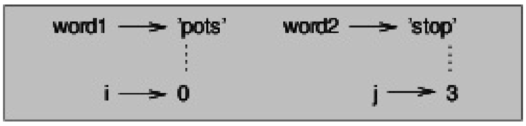


Figure 8.2: State diagram.

I took some license by arranging the variables in the frame and adding dotted lines to show that the values of i and j indicate characters in word1and word2. Starting with this diagram, run the program on paper, changing the values ofi and j during each iteration. Find and fix the second error in this function.

> 我通过设置变量框架中添加虚线表明，i和j的值显示在人物word1and word2拿许可证。

从这个图上运行的程序，文件，更改这些值I和J在每一次迭代过程。发现并解决此函数中的二次错误。

8.12 Glossary 术语列表

object: Something a variable can refer to. For now, you can use “object” and “value” interchangeably.

> 对象：一个值能够指代的东西。目前为止，你可以把对象和值暂且作为一码事来理解。

sequence: An ordered collection of values where each value is identified by an integer index.

> 序列：一系列值的有序排列，每一个值都有一个唯一的整数序号。

item: One of the values in a sequence.

> 元素：一列数值序列当中的一个值。

index: An integer value used to select an item in a sequence, such as a character in a string. In Python indices start from 0.

> 索引：一个整数值，用来指代一个序列中的特定一个元素，比如在字符串里面就指代一个字符。在 Python 里面索引从0开始计数。

slice: A part of a string specified by a range of indices.

> 切片：字符串的一部分，通过一个索引区间来取得。

empty string: A string with no characters and length 0, represented by two quotation marks.

> 空字符串：没有字符的字符串，长度为0，用两个单引号表示。

immutable: The property of a sequence whose items cannot be changed.

> 不可更改：一个序列中所有元素不能被改变的性质。

traverse: To iterate through the items in a sequence, performing a similar operation on each.

> 遍历：在一个序列中依次对每一个元素进行某种相似运算的过程。

search: A pattern of traversal that stops when it finds what it is looking for.

> 搜索：一种遍历的模式，找到要找的内容的时候就停止。

counter: A variable used to count something, usually initialized to zero and then incremented.

> 计数：一种用来统计某种东西数量的变量，一般初始化为0，然后逐次递增。

invocation: A statement that calls a method.

> 方法调用：调用方法的语句。

optional argument: A function or method argument that is not required.

> 可选参数：一个函数或者方法中有一些参数是可选的，非必需的。

8.13 Exercises 练习

Exercise 1 练习1

Read the documentation of the string methods at here. You might want to experiment with some of them to make sure you understand how they work. strip and replace are particularly useful.

> 阅读 这里关于字符串的文档。你也许会想要试试其中一些方法，来确保你理解它们的意义。比如 strip 和 replace 都特别有用。

The documentation uses a syntax that might be confusing. For example, in find(sub[, start[, end]]), the brackets indicate optional arguments. So sub is required, but start is optional, and if you include start, then end is optional.

> 文档的语法有可能不太好理解。比如在find 这个方法中，方括号表示了可选参数。所以 sub 是必须的参数，但 start 是可选的，如果你包含了 start，end 就是可选的了。

Exercise 2 练习2

There is a string method called count that is similar to the function in Section 8.7. Read the documentation of this method and write an invocation that counts the number of 'a's in 'banana'.

> 字符串有个方法叫 count，与咱们在8.7中写的 count 函数很相似。 阅读一下这个方法的文档，然后写一个调用这个方法的代码，统计一下 banana 这个单词中 a 出现的次数 。

Exercise 3 练习3

A string slice can take a third index that specifies the “step size;” that is, the number of spaces between successive characters. A step size of 2 means every other character; 3 means every third, etc.

> 字符串切片可以使用第三个索引，作为步长来使用；步长的意思就是取字符的间距。一个步长为2的意思就是每隔一个取一个字符；3的意思就是每次取第三个，以此类推。

```
>>> fruit = 'banana'
>>> fruit = 'banana'
>>> fruit[0:5:2]
>>> fruit[0:5:2]
'bnn'
```

A step size of -1 goes through the word backwards, so the slice [::-1]generates a reversed string. Use this idiom to write a one-line version of is_palindrome from Exercise 3.

> 步长如果为-1，意思就是倒序读取字符串，所以[::-1]这个切片就会生成一个逆序的字符串了。
>
> 使用这个方法把练习三当中的is_palindrome写成一个一行代码的版本。

Exercise 4 练习4

The following functions are all intended to check whether a string contains any lowercase letters, but at least some of them are wrong. For each function, describe what the function actually does (assuming that the parameter is a string).

> 下面这些函数都试图检查一个字符串是不是包含小写字母，但他们当中肯定有些是错的。描述一下每个函数真正的行为（假设参数是一个字符串）。

```
def any_lowercase1(s):
    for c in s:
        if c.islower():
            return True
        else:
            return False
def any_lowercase2(s):
    for c in s:
        if 'c'.islower():
            return 'True'
        else:
            return 'False'
def any_lowercase3(s):
    for c in s:
        flag = c.islower()
        return flag
def any_lowercase4(s):
    flag = False
        for c in s:
            flag = flag or c.islower()
            return flag
def any_lowercase5(s):
    for c in s:
        if not c.islower():
            return False
        return True
```

Exercise 5 练习5

A Caesar cypher is a weak form of encryption that involves “rotating” each letter by a fixed number of places. To rotate a letter means to shift it through the alphabet, wrapping around to the beginning if necessary, so ’A’ rotated by 3 is ’D’ and ’Z’ rotated by 1 is ’A’.

> 凯撒密码是一种简单的加密方法，用的方法是把每个字母进行特定数量的移位。对一个字母移位就是把它根据字母表的顺序来增减对应，如果到末尾位数不够就从开头算剩余的位数，『A』移位3就是『D』，而『Z』移位1就是『A』了。

To rotate a word, rotate each letter by the same amount. For example, “cheer” rotated by 7 is “jolly” and “melon” rotated by -10 is “cubed”. In the movie 2001: A Space Odyssey, the ship computer is called HAL, which is IBM rotated by -1.

> 要对一个词进行移位，要把每个字母都移动同样的数量。比如『cheer』这个单词移位7就是『jolly』，而『melon』移位-10就是『cubed』。在电影《2001 太空漫游》中，飞船的电脑叫 HAL，就是 IBM 移位-1。

Write a function called rotate_word that takes a string and an integer as parameters, and returns a new string that contains the letters from the original string rotated by the given amount.

> 写一个名叫 rotate_word 的函数，接收一个字符串和一个整形为参数，返回将源字符串移位该整数位得到的新字符串。

You might want to use the built-in function ord, which converts a character to a numeric code, and chr, which converts numeric codes to characters. Letters of the alphabet are encoded in alphabetical order, so for example:

> 你也许会用得上内置函数 ord，它把字符转换成数值代码，然后还有个 chr 是用来把数值代码转换成字符。字母表中的字母都被编译成跟字母表中同样的顺序了，所以如下所示：

```
>>> ord('c') - ord('a')
>>> ord('c') - ord('a')
2
```

Because 'c' is the two-eth letter of the alphabet. But beware: the numeric codes for upper case letters are different.

> c 是字母表中的第『2』个（译者注：从0开始数哈）的位置，所以上述结果是2。但注意：大写字母的数值代码是和小写的不一样的。

Potentially offensive jokes on the Internet are sometimes encoded in ROT13, which is a Caesar cypher with rotation 13. If you are not easily offended, find and decode some of them. Solution.

> 网上很多有冒犯意义的玩笑都是用 ROT13加密的，也就是移位13的凯撒密码。如果你不太介意，找一下这些密码解密一下吧。样例代码.

Chapter 9 Case study: word play 案例学习：单词游戏

This chapter presents the second case study, which involves solving word puzzles by searching for words that have certain properties. For example, we'll find the longest palindromes in English and search for words whose letters appear in alphabetical order. And I will present another program development plan: reduction to a previously-solved problem.

> 本章我们进行第二个案例学习，这一案例中涉及到了用搜索具有某些特征的单词来猜谜。比如，我们会发现英语中最长的回文词，然后搜索那些按照单词表顺序排列字母的单词。我还会给出一种新的程序开发计划：降低问题的复杂性和难度，还原到以前解决的问题。

9.1 Reading word lists 读取字符列表

For the exercises in this chapter we need a list of English words. There are lots of word lists available on the Web, but the one most suitable for our purpose is one of the word lists collected and contributed to the public domain by Grady Ward as part of the Moby lexicon project (see Here). It is a list of 113,809 official crosswords; that is, words that are considered valid in crossword puzzles and other word games. In the Moby collection, the filename is 113809of.fic; you can download a copy, with the simpler name words.txt, from Here.

> 本章练习中，咱们需要用一个英语词汇列表。网上有很多，不过最适合我们的列表并且是共有领域的，莫过于 Grady Ward这份词汇表，这是Moby词典计划的一部分（点击此链接访问详情）。这是一份113，809个公认的字谜表；也就是公认可以用于字谜游戏以及其他文字游戏的单词。在 Moby 的词汇项目中，该词表的文件名为113809of.fic；你可以下载一份副本，这里名字简化成 words.txt 了，下载地址在这里。

This file is in plain text, so you can open it with a text editor, but you can also read it from Python. The built-in function open takes the name of the file as a parameter and returns a file object you can use to read the file.

> 这个文件就是纯文本，所以你可以用文本编辑器打开一下，不过也可以用 Python 来读取。Python 内置了一个叫open 的函数，接收文件名做参数，返回一个文件对象，你可以用它来读取文件。

```
>>> fin = open('words.txt')
>>> fin = open('words.txt')
```

fin is a common name for a file object used for input. The file object provides several methods for reading, including readline, which reads characters from the file until it gets to a newline and returns the result as a string:

> fin 是一个用来表示输入的文件的常用名字。这个文件对象提供了好几种读取的方法，包括逐行读取，这种方法是读取文本中的一整行直到结尾，然后把读取的内容作为字符串返回：

```
>>> fin.readline()
>>> fin.readline()
'aa\r\n'
```

The first word in this particular list is “aa”, which is a kind of lava. The sequence \r\n represents two whitespace characters, a carriage return and a newline, that separate this word from the next. The file object keeps track of where it is in the file, so if you call readline again, you get the next word:

> 这一列当中的第一个词就是『aa』了，这是一种熔岩（译者注：“aa”是夏威夷词汇，音“阿阿”，用来描述表面粗糙的熔岩流。译者本人就是地学专业的，都很少接触这个词，本教材作者真博学啊）。后面跟着的\r\n 的意思代表着有两个转义字符，一个是回车，一个是换行，这样把这个单词从下一个单词分隔开来。
>
> 文件对象会记录这个单词在源文件中的位置，所以下次你再调用 readline 的时候，就能得到下一个词了：

```
>>> fin.readline()
>>> fin.readline()
'aah\r\n'
```

The next word is “aah”, which is a perfectly legitimate word, so stop looking at me like that. Or, if it’s the whitespace that’s bothering you, we can get rid of it with the string method strip:

> 下一个词是『aah』，这完全是一个正规的词汇，不要怪异眼神看我哦。另外如果转义字符让你很烦，咱们可以稍加修改来去掉它，用字符串的 strip 方法即可：

```
>>> line = fin.readline()
>>> line = fin.readline()
>>> word = line.strip()
>>> word = line.strip()
>>> word
>>> word
'aahed'
```

You can also use a file object as part of a for loop. This program reads words.txt and prints each word, one per line:

> 在 for 循环中也可以使用文件对象。下面的这个程序读取整个 words.txt 文件，然后每行输出一个词：

```
fin = open('words.txt')
    for line in fin:
        word = line.strip()
        print(word)
```

9.2 Exercises 练习

There are solutions to these exercises in the next section. You should at least attempt each one before you read the solutions.

> 下面这些练习都有样例代码。不过你最好在看答案之前先自己对每个练习都尝试一下。

Exercise 1 练习1

Write a program that reads words.txt and prints only the words with more than 20 characters (not counting whitespace).

> 写一个程序读取 words.txt，然后只输出超过20个字母长度的词（这个长度不包括转义字符）。

Exercise 2 练习2

In 1939 Ernest Vincent Wright published a 50,000 word novel called Gadsby that does not contain the letter “e”. Since “e” is the most common letter in English, that’s not easy to do.

> 在1939年，作家厄尔尼斯特·文森特·莱特曾经写过一篇5万字的小说《葛士比》，里面没有一个字母e。因为在英语中 e 是用的次数最多的字母，所以这很不容易的。

In fact, it is difficult to construct a solitary thought without using that most common symbol. It is slow going at first, but with caution and hours of training you can gradually gain facility.

> 事实上，不使用最常见的字符，都很难想出一个简单的想法。一开始很慢，不过仔细一些，经过几个小时的训练之后，你就逐渐能做到了。

All right, I’ll stop now. Write a function called has_no_e that returns True if the given word doesn’t have the letter “e” in it.

> 好了，我不扯淡了。
>
> 写一个名字叫做 has_no_e 的函数，如果给定词汇不含有 e 就返回真，否则为假。

Modify your program from the previous section to print only the words that have no “e” and compute the percentage of the words in the list that have no “e”.

> 修改一下上一节的程序代码，让它只打印单词表中没有 e 的词汇，并且统计一下这些词汇在总数中的百分比例。

Exercise 3 练习3

Write a function named avoids that takes a word and a string of forbidden letters, and that returns True if the word doesn’t use any of the forbidden letters. Modify your program to prompt the user to enter a string of forbidden letters and then print the number of words that don’t contain any of them. Can you find a combination of 5 forbidden letters that excludes the smallest number of words?

> 写一个名叫 avoids 的函数，接收一个单词和一个禁用字母组合的字符串，如果单词不含有该字符串中的任何字母，就返回真。
>
> 修改一下程序代码，提示用户输入一个禁用字母组合的字符串，然后输入不含有这些字母的单词数目。你能找到5个被禁用字母组合，排除单词数最少吗？

Exercise 4 练习4

Write a function named uses_only that takes a word and a string of letters, and that returns True if the word contains only letters in the list. Can you make a sentence using only the letters acefhlo? Other than “Hoe alfalfa”?

> 写一个名叫uses_only的函数，接收一个单词和一个字母字符串，如果单词仅包含该字符串中的字母，就返回真。你能仅仅用 acefhlo 这几个字母造句子么？或者试试『Hoe alfalfa』？

Exercise 5 练习5

Write a function named uses_all that takes a word and a string of required letters, and that returns True if the word uses all the required letters at least once. How many words are there that use all the vowels aeiou? How about aeiouy?

> 写一个名字叫uses_all的函数，接收一个单词和一个必需字母组合的字符串，如果单词对必需字母组合中的字母至少都用了一次就返回真。有多少单词都用到了所有的元音字母aeiou？aeiouy的呢？

Exercise 6 练习6

Write a function called is_abecedarian that returns True if the letters in a word appear in alphabetical order (double letters are ok). How many abecedarian words are there?

> 写一个名字叫is_abecedarian的函数，如果单词中所有字母都是按照字母表顺序出现就返回真（重叠字母也是允许的）。有多少这样的单词？

9.3 Search 搜索

All of the exercises in the previous section have something in common; they can be solved with the search pattern we saw in Section 8.6. The simplest example is:

> 刚刚的那些练习都有一些相似之处：都可以用我们在8.6学过的搜索来解决。下面是一个最简化的例子：

```
def has_no_e(word):
    for letter in word:
        if letter == 'e':
            return False
    return True
```

The for loop traverses the characters in word. If we find the letter “e”, we can immediately return False; otherwise we have to go to the next letter. If we exit the loop normally, that means we didn’t find an “e”, so we return True.

> 这个 for 循环遍历了单词的所有字母。如果找到了字母e，就立即返回假；否则就到下一个字母。如果正常退出了循环，意味着我们没找到 e，就返回真。

You could write this function more concisely using the in operator, but I started with this version because it demonstrates the logic of the search pattern. avoids is a more general version of has_no_e but it has the same structure:

> 你可以使用 in 运算符，把这个函数写得更精简，我之所以用一个稍显麻烦的版本，是为了说明搜索模式的逻辑过程。
>
> avoids 是一个更通用版本的has_no_e函数的实现，它的结构是一样的：

```
def avoids(word, forbidden):
    for letter in word:
        if letter in forbidden:
            return False
    return True
```

We can return False as soon as we find a forbidden letter; if we get to the end of the loop, we return True. uses_only is similar except that the sense of the condition is reversed:

> 只要找到了禁用字母就可以立即返回假；如果运行到了循环末尾，就返回真。
>
> uses_only与之相似，无非是条件与之相反了而已。

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

Instead of a list of forbidden letters, we have a list of available letters. If we find a letter in word that is not in available, we can return False. uses_all is similar except that we reverse the role of the word and the string of letters:

> 这次不是有一个禁用字母列表，我们这次用一个可用字母列表。如果在单词中发现不在可用字母列表中的，就返回假了。
>
> uses_all这个函数与之也相似，不过我们转换了单词和字母字符串的角色：

```
def uses_all(word, required):
    for letter in required:
        if letter not in word:
            return False
    return True
```

Instead of traversing the letters in word, the loop traverses the required letters. If any of the required letters do not appear in the word, we can return False. If you were really thinking like a computer scientist, you would have recognized that uses_all was an instance of a previously-solved problem, and you would have written:

> 这次并没有遍历单词中的所有字母，循环遍历了所有指定的字母。如果有任何指定字母没有在单词中出新啊，就返回假。如果你已经像计算机科学家一样思考了，你就应该已经发现了uses_all是对之前我们解决过问题的一个实例，你已经写过这个代码了：

```
def uses_all(word, required):
    return uses_only(required, word)
```

This is an example of a program development plan called reduction to a previously-solved problem, which means that you recognize the problem you are working on as an instance of a solved problem and apply an existing solution.

> 这就是一种新的程序开发规划模式，就是降低问题的复杂性和难度，还原到以前解决的问题，意思是你发现正在面对的问题是之前解决过的问题的一个实例，就可以用已经存在的方案来解决。

9.4 Looping with indices 带索引循环

I wrote the functions in the previous section with for loops because I only needed the characters in the strings; I didn't have to do anything with the indices. For is_abecedarian we have to compare adjacent letters, which is a little tricky with a for loop:

> 上面的章节中我写了各种用 for 循环的函数，因为当时只需要字符串中的字符；这就不需要理会索引。
>
> 但is_abecedarian这个函数中，我们需要对比临近的两个字母，所以用 for 循环就不那么好写了：

```
def is_abecedarian(word):
    previous = word[0]
    for c in word:
        if c < previous:
            return False
        previous = c
    return True
```

An alternative is to use recursion:

> 一种很好的替代思路就是使用递归：

```
def is_abecedarian(word):
    if len(word) <= 1:
        return True
    if word[0] > word[1]:
        return False
    return is_abecedarian(word[1:])
```

Another option is to use a while loop:

> 另外一种方法是用 while 循环：

```
def is_abecedarian(word):
    i = 0
    while i < len(word)-1:
        if word[i+1] < word[i]:
        return False
        i = i+1
    return True
```

The loop starts at i=0 and ends when i=len(word)-1. Each time through the loop, it compares the ith character (which you can think of as the current character) to the i+1th character (which you can think of as the next).

> 循环开始于 i 等于0，然后在 i 等于len(word)-1的时候结束。每次通过循环的时候，都对比第 i 个字符（你可以就当是当前字符）与第 i+1个字符（就当作下一个字符）。

If the next character is less than (alphabetically before) the current one, then we have discovered a break in the abecedarian trend, and we return False.

> 如果下一个字符比当前字符小（字母表排列顺序在当前字符前面），我们就发现这个不符合字母表顺序了，跳出返回假就可以了。

If we get to the end of the loop without finding a fault, then the word passes the test. To convince yourself that the loop ends correctly, consider an example like 'flossy'. The length of the word is 6, so the last time the loop runs is when i is 4, which is the index of the second-to-last character. On the last iteration, it compares the second-to-last character to the last, which is what we want.

> 如果一直到循环结尾都没有发现问题，这个词就通过检验了。为了确信循环正确结束了，可以拿单词『flossy』作为例子来试试。单词长度是6，所以循环终止的时候 i 应该是4，也就是倒数第二个位置。在最后一次循环中，比较的是倒数第二个和最后一个字母，这正是符合我们设计的。

Here is a version of is_palindrome (see Exercise 3) that uses two indices; one starts at the beginning and goes up; the other starts at the end and goes down.

> 下面这个是练习3的is_palindrome的一个版本，使用了两个索引；一个从头开始一直到结尾；另外一个从末尾开始逆序进行。

```
def is_palindrome(word):
    i = 0
    j = len(word)-1
    while i<j:
        if word[i] != word[j]:
            return False
        i = i+1
        j = j-1
    return True
```

Or we could reduce to a previously-solved problem and write:

> 或者我们可以把问题解构成之前解决过的样式，然后这样写：

```
def is_palindrome(word):
    return is_reverse(word, word)
```

Using is_reverse from Section 8.11.

> 这里的is_reverse这个函数在第8章第11节讲过哈。

9.5 Debugging 调试

Testing programs is hard. The functions in this chapter are relatively easy to test because you can check the results by hand. Even so, it is somewhere between difficult and impossible to choose a set of words that test for all possible errors.

> 测试程序很难的。本章的函数相对来说还算容易测试，因为你可以手动计算来检验结果。即便如此，选择一系列单词然后检测所有可能的错误，可能不仅是做起来困难，甚至都是不可能完成的任务。

Taking has_no_e as an example, there are two obvious cases to check: words that have an 'e' should return False, and words that don't should return True. You should have no trouble coming up with one of each.

> 比如以has_no_e为例，有两种情况用来检查：有 e 的单词应该返回假，不包含 e 的单词要返回真。你自己想出几个这样的单词来检验一下并不难。

Within each case, there are some less obvious subcases. Among the words that have an "e", you should test words with an "e" at the beginning, the end, and somewhere in the middle. You should test long words, short words, and very short words, like the empty string. The empty string is an example of a special case, which is one of the non-obvious cases where errors often lurk.

> 在每个分支内，有一些不那么清晰的次级分支。在那些有 e 的单词中，你还要检测单词中的 e 是在开头结尾还是中间位置。你得试试长词、短词，甚至特别短的词，比如空字符串。空字符串是一个典型特例，这个情况很容易被忽视而成为潜伏的隐患。
>
> （译者注：我知道，这段翻译的简直就是 shit，但是没办法，我这会眼睛特别疼，思路不太清楚，另外这几个练习也不是很难，大家很容易自己搞定。）

In addition to the test cases you generate, you can also test your program with a word list like words.txt. By scanning the output, you might be able to catch errors, but be careful: you might catch one kind of error (words that should not be included, but are) and not another (words that should be included, but aren't).

> 除了你自己设计的测试案例之外，你也可以用一个单词列表比如 words.txt 之类的来测试一下你的程序。通过扫描一下输出内容，你也许能够发现错误的地方，但一定要小心：你有可能发现某一种特定错误，但忽略了另外一个，比如包含了不应该包含的单词，但很难发现应该包含但遗漏了单词的情况。

In general, testing can help you find bugs, but it is not easy to generate a good set of test cases, and even if you do, you can't be sure your program is correct. According to a legendary computer scientist: Program testing can be used to show the presence of bugs, but never to show their absence! — Edsger W. Dijkstra

> 总的来说，测试程序能帮助你找到错误地方，但很难找到一系列特别好的测试案例，或者即便你找了很多案例来测试，也不能确保程序就是正确的。一位传说级别的计算机科学家说：
>
> 程序测试可以用来表明 bug 的存在，但永远不能表明 bug 不存在。
>
> — Edsger W. Dijkstra

9.6 Glossary 术语列表

file object: A value that represents an open file.

> 文件对象：代表了一份打开的文件的值。

reduction to a previously-solved problem: A way of solving a problem by expressing it as an instance of a previously-solved problem.

> 降低问题的复杂性和难度，还原到以前解决的问题：一种解决问题的方法，把问题表达成过去解决过问题的一个特例。

special case: A test case that is a typical or non-obvious (and less likely to be handled correctly).

> 特殊案例：很典型或者不明显的测试用的案例，往往都很不容易正确处理。

9.7 Exercises 练习

Exercise 7 练习7

This questionis based on a Puzzler that was broadcast on the radio program Car Talk : Give me a word with three consecutive double letters. I'll give you a couple of words that almost qualify, but don't. For example, the word committee, c-o-m-m-i-t-t-e-e. It would be great except for the 'i' that sneaks in there. Or Mississippi: M-i-s-s-i-s-s-i-p-p-i. If you could take out those i's it would work. But there is a word that has three consecutive pairs of letters and to the best of my knowledge this may be the only word. Of course there are probably 500 more but I can only think of one. What is the word? Write a program to find it. Solution

> 这个问题基于一个谜语，这个谜语在广播节目 Car Talk 上面播放过：
>
> 给我一个有三个连续双字母的单词。我会给你一对基本符合的单词，但并不符合。例如， committee 这个单词，C O M M I T E。如果不是有单独的一个 i 在里面，就基本完美了，或者Mississippi 这个词：M I S I S I P I。如果把这些个 i 都去掉就好了。但有一个词正好是三个重叠字母，而且据我所知这个词可能是唯一一个这样的词。当然有有可能这种单词有五百多个呢，但我只能想到一个。是哪个词呢？写个程序来找一下这个词吧。
>
> 样例代码

Exercise 8 练习8

Here's another Car Talk Puzzler : "I was driving on the highway the other day and I happened to notice my odometer. Like most odometers, it shows six digits, in whole miles only. So, if my car had 300,000 miles, for example, I'd see 3-0-0-0-0-0. "Now, what I saw that day was very interesting. I noticed that the last 4 digits were palindromic; that is, they read the same forward as backward. For example, 5-4-4-5 is a palindrome, so my odometer could have read 3-1-5-4-4-5. "One mile later, the last 5 numbers were palindromic. For example, it could have read 3-6-5-4-5-6. One mile after that, the middle 4 out of 6 numbers were palindromic. And you ready for this? One mile later, all 6 were palindromic! "The question is, what was on the odometer when I first looked?" Write a Python program that tests all the six-digit numbers and prints any numbers that satisfy these requirements. Solution

> 这个又是一个 Car Talk 谜语：
>
> 有一天我在高速路上开着车，碰巧看了眼里程表。和大多数里程表一样，是六位数字的，单位是英里。加入我的车跑过300，000英里了，看到的结果就是3-0-0-0-0-0.
>
> 我那天看到的很有趣，我看到后四位是回文的；就是说后四位正序和逆序读是一样的。例如5-4-4-5就是一个回文数，所以我的里程表可能读书就是3-1-5-4-4-5.
>
> 过了一英里之后，后五位数字是回文的了。举个例子，可能读书就是3-6-5-4-5-6。又过了一英里，六个数字中间的数字是回文数了。准备好更复杂的了么？又过了一英里，整个六位数都是回文的了！
>
> 那么问题俩了：我最开始看到的里程表的度数应该是多少？
>
> 写个 Python 程序来检测一下所有的六位数，然后输出一下满足这些要求的数字。 样例代码

Exercise 9 练习9

Here's another Car Talk Puzzler you can solve with a search : "Recently I had a visit with my mom and we realized that the two digits that make up my age when reversed resulted in her age. For example, if she's 73, I'm 37. We wondered how often this has happened over the years but we got sidetracked with other topics and we never came up with an answer. "When I got home I figured out that the digits of our ages have been reversible six times so far. I also figured out that if we're lucky it would happen again in a few years, and if we're really lucky it would happen one more time after that. In other words, it would have happened 8 times over all. So the question is, how old am I now?" Write a Python program that searches for solutions to this Puzzler. Hint: you might find the string method zfill useful. Solution

> 这个又是一个 Car Talk 谜语，你可以用搜索来解决：
>
> 最近我看忘了妈妈，然后我们发现我的年龄反过来正好是她的年龄。例如，假如他是73岁，我就是37岁了。我们好奇这种情况发生多少次，但中间又开了话题，没有想出来这个问题的答案。
>
> 我回家之后，我发现到目前位置我们的年龄互为逆序已经是六次了，我还发现如果我们幸运的话过几年又会有一次，如果我们特别幸运，就还会再有一次这样情况。换句话说，就是总共能有八次。那么问题来了：我现在多大了？
>
> 写一个 Python 程序，搜索一下这个谜语的解。提示一下：你可能发现字符串的 zfill 方法很有用哦。
>
> 样例代码

Chapter 10 Lists 列表

This chapter presents one of Python's most useful built-in types, lists. You will also learn more about objects and what can happen when you have more than one name for the same object.

> 本章讲述的是 Python 里面最有用的一种内置类型：列表。你还能在本章中学到更多关于类的内容，你还会看到如果同一个对象有多个名字会有什么情况发生。

10.1 A list is a sequence 列表即序列

Like a string, a list is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in a list are called elements or sometimes .

> 和字符串差不多，列表是一系列的数值的序列。在字符串里面，这些值是字符；在列表里面，这些值可以是任意类型的。一个列表中的值一般叫做列表的元素，有时候也叫列表项。

There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([and]):

> 创建一个新的列表有好几种方法；最简单的方法就是把列表的元素用方括号包含起来：

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

The first example is a list of four integers. The second is a list of three strings. The elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and (lo!) another list:

> 第一个例子建立了一个由四个整形变量组成的列表。第二个是一个由三个字符串组成的列表。

```
['spam', 2.0, 5, [10, 20]]
```

A list within another list is nested. A list that contains no elements is called an empty list; you can create one with empty brackets, []. As you might expect, you can assign list values to variables:

> 列表内部可以包含一个列表作为元素，这种包含列表的列表也叫做网状列表。
>
> 不含有任何元素的列表叫做空列表；可以就用空的方括号来建立一个。
>
> 你估计也会预料到，列表的值可以赋给变量：

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [42, 123]
>>> numbers = [42, 123]
>>> empty = []
>>> empty = []
>>> print(cheeses, numbers, empty)
>>> print(cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda']
[42, 123]
[]
```

10.2 Lists are mutable 列表元素可修改

The syntax for accessing the elements of a list is the same as for accessing the characters of a string—the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0:

> 读取列表元素的语法就如同读取字符串中的字符一样-用方括号运算符就可以了。方括号内的数字用来确定索引位置。一定要记住，Python 是从零开始计数的：

```
>>> cheeses[0]
>>> cheeses[0]
'Cheddar'
```

Unlike strings, lists are mutable. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

> 和字符串不同的是，列表是可以修改的。方括号运算符放到一个赋值语句的等号左侧的时候，就会把对应位置的列表元素重新赋值。

```
>>> numbers = [42, 123]
>>> numbers = [42, 123]
>>> numbers[1] = 5
>>> numbers[1] = 5
>>> numbers
>>> numbers
[42, 5]
```

The one-eth element of numbers, which used to be 123, is now 5. Figure 10.1 shows the state diagram for cheeses, numbers and empty:

> 列表 numbers 的第『1』个元素之前是123，现在被改为5了。
>
> 图10.1展示了 cheeses、numbers 和空列表的状态图：

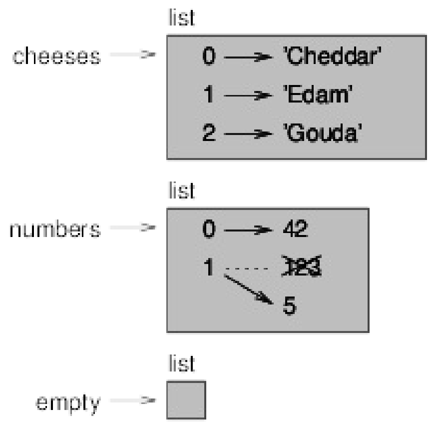


Figure 10.1: State diagram.

Lists are represented by boxes with the word “list” outside and the elements of the list inside. cheeses refers to a list with three elements indexed 0, 1 and 2.numbers contains two elements; the diagram shows that the value of the second element has been reassigned from 123 to 5. empty refers to a list with no elements.

> 这三个列表都用标记了『list』的三个小盒子表示，盒子外面的是列表的名字，盒子内的就是列表元素了。cheese 是一个有0，1，2三个元素的列表。numbers 包含两个元素；图示表明第二个元素的值从123被重新赋值为5。empty 是一个不包含元素的空列表。

List indices work the same way as string indices:

> 列表的索引和字符串的索引的格式是一样的：

• Any integer expression can be used as an index.

> 任意的一个整型表达式，都可以用来作为索引编号。

• If you try to read or write an element that does not exist, you get an IndexError.

> 如果你试图读取或者写入一个不存在的列表元素，你就会得到一个索引错误 IndexError。

• If an index has a negative value, it counts backward from the end of the list.

> 如果一个索引是负值，意味着是从列表末尾向前倒序计数查找相对应的位置。

The in operator also works on lists.

> 在列表中也可以使用 in 运算符。

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
>>> 'Brie' in cheeses
False
```

10.3 Traversing a list 遍历一个列表

The most common way to traverse the elements of a list is with a for loop. The syntax is the same as for strings:

> 遍历一个列表中所有元素的最常用的办法就是 for 循环了。这种 for 循环和我们在遍历一个字符串的时候用的是一样的的语法：

```
for cheese in cheeses:
    print(cheese)
```

This works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to do that is to combine the built-in functions range and len:

> 如果你只是要显示一下列表元素，上面这个代码就够用了。但如果你还想写入或者更新这些元素，你还是需要用索引。一般来说，这需要把两个内置函数 range 和 len 结合起来使用：

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

This loop traverses the list and updates each element. len returns the number of elements in the list. range returns a list of indices from 0 to n−1, where n is the length of the list. Each time through the loop i gets the index of the next element. The assignment statement in the body uses i to read the old value of the element and to assign the new value.

> 这个循环遍历了列表，然后对每个元素都进行了更新。len 这个函数返回的是列表中元素的数量。range 返回的是列表的一系列索引，从0到 n-1，n 就是整个列表的长度了。每次循环的时候，i 都会得到下一个元素的索引值。在循环体内部的赋值语句每次都通过 i 作为索引来读该元素的旧值，进行修改然后赋新值给该元素。

A for loop over an empty list never runs the body:

> 空列表的 for 循环中，循环体是永远不会运行的：

```
for x in []:
    print('This never happens.')
```

Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

> 尽管列表中可以办好另外一个列表，但这种网状的分支列表依然只会被算作一个元素。所以下面这个列表的长度是4：

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

10.4 List operations 列表运算符

The + operator concatenates lists:

> 加号+运算符可以把列表拼接在一起：

```
>>> a = [1, 2, 3]
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> b = [4, 5, 6]
>>> c = a + b
>>> c = a + b
>>> c
>>> c
[1, 2, 3, 4, 5, 6]
```

The * operator repeats a list a given number of times:

> 星号*运算符可以将列表重复指定的次数：

```
>>> [0] * 4
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The first example repeats [0] four times. The second example repeats the list[1, 2, 3] three times.

> 第一个例子中，[0]这个列表被重复四次。第二个例子把列表[1,2,3]重复了三次。

10.5 List slices 列表切片

The slice operator also works on lists:

> 切片操作符也可以用到列表上：

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
>>> t[1:3]
['b', 'c']
>>> t[:4]
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
>>> t[3:]
['d', 'e', 'f']
```

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list.

> 在切片运算中，如果你省略了第一个索引，切片就会从头开始。如果省略了第二个，切片就会一直走到末尾。所以如果你把两个都省略了，这个切片就是整个列表的一个复制了。

```
>>> t[:]
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Since lists are mutable, it is often useful to make a copy before performing operations that modify lists. A slice operator on the left side of an assignment can update multiple elements:

> 因为列表是可以修改的，所以在进行运算修改列表之前，做个复制来备份经常是很有必要的。
>
> 切片运算符放到赋值语句中等号左边的时候可以对多个元素进行更新：

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> t[1:3] = ['x', 'y']
>>> t
>>> t
['a', 'x', 'y', 'd', 'e', 'f']
```

10.6 List methods 列表的方法

Python provides methods that operate on lists. For example, append adds a new element to the end of a list:

> Python 为操作列表提供了很多方法。比如，append 就可以在列表末尾添加一个新的元素：

```
>>> t = ['a', 'b', 'c']
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> t.append('d')
>>> t
>>> t
['a', 'b', 'c', 'd']
```

extend takes a list as an argument and appends all of the elements:

> extend 使用另一个列表做参数，然后把所有的元素添加到一个列表上。

```
>>> t1 = ['a', 'b', 'c']
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> t1.extend(t2)
>>> t1
>>> t1
['a', 'b', 'c', 'd', 'e']
```

This example leaves t2 unmodified.

> 上面这个例子中，t2是没有修改过的。

sort arranges the elements of the list from low to high:

> sort 把列表中的元素从低到高（译者注：下面的例子中是按照 ASCII 码的大小从小到大）排列：

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> t.sort()
>>> t
>>> t
['a', 'b', 'c', 'd', 'e']
```

Most list methods are void; they modify the list and return None. If you accidentally write t = t.sort(), you will be disappointed with the result.

> 大多数列表的方法都是无返回值的；这些方法都对列表进行修改，返回的是空。如果你不小心写出了一个 t=t.sort()，得到的结果恐怕让你很失望。

10.7 Map, filter and reduce 列表中最重要的三种运算

To add up all the numbers in a list, you can use a loop like this:

> 要得到列表中所有值的综合，你可以用下面这样的一个循环来实现：

```
def add_all(t):
    total = 0
    for x in t:
        total += x
    return total
```

total is initialized to 0. Each time through the loop, x gets one element from the list. The += operator provides a short way to update a variable. This augmented assignment statement, total += x is equivalent to total = total + x .

> total 的初始值为0。每次循环的时候，x 都得到列表中一个元素的值。+=这个运算符是更新变量的一种简写。这个运算符是扩展了赋值语句，total += x 就等同于 total = total +x 。

As the loop runs, total accumulates the sum of the elements; a variable used this way is sometimes called an accumulator.

> 随着循环的运行，total 积累了所有元素的综合；这种变量也叫做累（三声）加器。

Adding up the elements of a list is such a common operation that Python provides it as a built-in function, sum:

> 把列表中所有元素加起来是一种很常用的运算，所以 Python 提供了内置的函数 sum：

```
>>> t = [1, 2, 3]
>>> t = [1, 2, 3]
>>> sum(t)
>>> sum(t)
6
```

An operation like this that combines a sequence of elements into a single value is sometimes called reduce.

> 把一系列列表元素组合成一个单值的运算，也叫做 reduce（这个单词是缩减的意思）。

Sometimes you want to traverse one list while building another. For example, the following function takes a list of strings and returns a new list that contains capitalized strings:

> 有时候建立一个列表需要遍历另一个列表。比如下面的这个函数就接收一个字符串列表，将所有字符串变为大写字母组成的，然后返回一个这些大写字符串组成的新列表：

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

res is initialized with an empty list; each time through the loop, we append the next element. So res is another kind of accumulator.

> res 的初始值为一个空列表；每次循环的时候，我们都把下一个元素用 append 方法拼接上去。所以 res 也算是另外一种累加器了。

An operation like capitalize_all is sometimes called a map because it “maps” a function (in this case the method capitalize) onto each of the elements in a sequence.

> 像上面这个capitalize_all 的运算也叫做一个 map（单词意思为地图），因为这种预算将某一函数（该例子中是 capitalize 这个方法）应用到一个序列中的每个元素上。

Another common operation is to select some of the elements from a list and return a sublist. For example, the following function takes a list of strings and returns a list that contains only the uppercase strings:

> 另外一种常见运算是从列表中选取一些元素，然后返回一个次级列表。比如，下面的函数接收一个字符串列表，然后返回一个其中只包含大写字母的字符串所组成的列表：

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

isupper is a string method that returns True if the string contains only upper case letters.

> isupper 是一个字符串方法，如果字符串中只包含大写字母就会返回真。

An operation like only_upper is called a filter because it selects some of the elements and filters out the others.

> only_upper 这样的运算也叫 filter（过滤器的意思），因为这种运算筛选出特定的元素，过滤掉其他的。

Most common list operations can be expressed as a combination of map, filter and reduce.

> 常用的列表运算都可以表示成 map、filter 以及 reduce 的组合。

10.8 Deleting elements 删除元素

There are several ways to delete elements from a list. If you know the index of the element you want, you can use pop:

> 从一个列表中删除元素有几种方法。如果你知道你要删除元素的索引，你就可以用 pop 这个方法来实现：

```
>>> t = ['a', 'b', 'c']
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> x = t.pop(1)
>>> t
>>> t
['a', 'c']
>>> x
>>> x
'b'
```

pop modifies the list and returns the element that was removed. If you don't provide an index, it deletes and returns the last element.

> pop 修改列表，然后返回删除的元素。如果你不指定一个索引位置，pop 就会删除和返回最后一个元素。

If you don't need the removed value, you can use the del operator:

> 如果你不需要删掉的值了，你可以用 del 运算符来实现：

```
>>> t = ['a', 'b', 'c']
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> del t[1]
>>> t
>>> t
['a', 'c']
```

If you know the element you want to remove (but not the index), you can use remove:

> 如果你知道你要删除的元素值，但不知道索引位置，你可以使用 remove 这个方法：

```
>>> t = ['a', 'b', 'c']
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> t.remove('b')
>>> t
>>> t
['a', 'c']
```

The return value from remove is None.

> remove 的返回值是空。

To remove more than one element, you can use del with a slice index:

> 要删除更多元素，可以使用 del 和切片索引：

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> del t[1:5]
>>> t
>>> t
['a', 'f']
```

As usual, the slice selects all the elements up to but not including the second index.

> 和之前我们看到的一样，切片是含头不含尾，上面这个例子中从第『1』到第『5』个都被切片所选中，但包含开头的第『1』而不包含末尾的第『5』个元素。

10.9 Lists and strings 列表和字符串

A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string. To convert from a string to a list of characters, you can use list:

> 字符串是一系列字符的序列，而李白是一系列值的序列，但一个由字符组成的列表是不同于字符串的。要把一个字符串转换成字符列表，你可以用 list 这个函数：

```
>>> s = 'spam'
>>> s = 'spam'
>>> t = list(s)
>>> t = list(s)
>>> t
>>> t
['s', 'p', 'a', 'm']
```

Because list is the name of a built-in function, you should avoid using it as a variable name. I also avoid l because it looks too much like 1. So that's why I use t.

> list 是一个内置函数的名字了，所以你应该避免用它来作为变量名。我还建议应该尽量少用 l，因为有的字体下，l 和1看着很难区分。所以我都用了 t。

The list function breaks a string into individual letters. If you want to break a string into words, you can use the split method:

> list 这个函数将一个字符串分开成一个个字母。如果你想把字符串切分成一个个单词，你可以用 split 这个方法：

```
>>> s = 'pining for the fjords'
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> t = s.split()
>>> t
>>> t
['pining', 'for', 'the', 'fjords']
```

An optional argument called a delimiter specifies which characters to use as word boundaries. The following example uses a hyphen as a delimiter:

> 可选的参数是定界符，是用来确定单词边界的。下面这个例子中就是把连接号『-』作为定界符：

```
>>> s = 'spam-spam-spam'
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> delimiter = '-'
>>> t = s.split(delimiter)
>>> t = s.split(delimiter)
>>> t
>>> t
['spam', 'spam', 'spam']
```

join is the inverse of split. It takes a list of strings and concatenates the elements. join is a string method, so you have to invoke it on the delimiter and pass the list as a parameter:

> join 是与 split 功能相反的一个方法。它接收一个字符串列表，然后把所有元素拼接到一起。join 是一个字符串方法，所以必须把 join 放到定界符后面来调用，并且传递一个列表作为参数：

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> delimiter = ' '
>>> s = delimiter.join(t)
>>> s = delimiter.join(t)
>>> s
>>> s
'pining for the fjords'
```

In this case the delimiter is a space character, so join puts a space between words. To concatenate strings without spaces, you can use the empty string,'', as a delimiter.

> 上面这个例子中，定界符是一个空格字符，所以 join 就在单词只见放一个空格。要想把字符聚集到一切而不要空格，你就可以用空字符串''作为一个定界符了。

10.10 Objects and values 对象和值

If we run these assignment statements:

> 如果我们运行下面这种赋值语句：

```
a = 'banana'
b = 'banana'
```

We know that a and b both refer to a string, but we don’t know whether they refer to the same string. There are two possible states, shown in Figure 10.2.

> 我们知道了 a 和 b 都是字符串，但我们不之道他们到底是不是同一个字符串。这就有可能有两种状态，如图10.2所示。

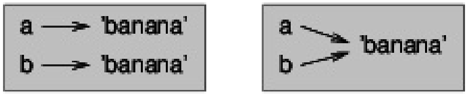


Figure 10.2: State diagram.

In one case, a and b refer to two different objects that have the same value. In the second case, they refer to the same object. To check whether two variables refer to the same object, you can use the is operator.

> 在第一种情况中，a 和 b 指向两个不同的对象，这两个对象有相同的值。在第二个情况下，a 和 b 都指向同一个对象。
>
> 要检查两个变量是否指向的是同一个对象，可以用 is 运算符。

```
>>> a = 'banana'
>>> a = 'banana'
>>> b = 'banana'
>>> b = 'banana'
>>> a is b
>>> a is b
True
```

In this example, Python only created one string object, and both a and b refer to it. But when you create two lists, you get two objects:

> 在这个例子中，Python 只建立了一个字符串对象，然后 a 和 b 都指向它。但当你建立两个列表的时候，你得到的就是两个对象了：

```
>>> a = [1, 2, 3]
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
>>> a is b
False
```

So the state diagram looks like Figure 10.3.

> 所以这时候的状态图就应该如图10.3所示的样子了。

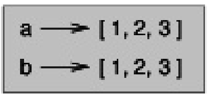


Figure 10.3: State diagram.

In this case we would say that the two lists are equivalent, because they have the same elements, but not identical, because they are not the same object. If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.

> 在这个情况下，我们可以说两个列表是相等的，因为它们有相同的元素，但它们不是同一个列表，因为他们并不是同一个对象。如果两个对象是同一个对象，那它们必然是相等的，但如果它们相等，却未必是同一个对象。
>
> （译者注：同一 是 相等 的充分条件，相等 是 同一 的必要条件，仅此而已。）

Until now, we have been using “object” and “value” interchangeably, but it is more precise to say that an object has a value. If you evaluate [1, 2, 3], you get a list object whose value is a sequence of integers. If another list has the same elements, we say it has the same value, but it is not the same object.

> 目前位置，我们一直把『对象』和『值』来随意交换使用，但实际上更确切的说法是一个对象拥有一个值。如果你计算[1,2,3]，你得到一个列表对象，整个列表对象的整个值是一个整数序列。如果另外一个列表有相同的元素，我们称之含有相同的值，但并不是相同的对象。

10.11 Aliasing 别名

If a refers to an object and you assign b = a, then both variables refer to the same object:

> 如果 a 是一个对象了，然后你赋值 b=a，那么这两个变量都指向同一个对象：

```
>>> a = [1, 2, 3]
>>> a = [1, 2, 3]
>>> b = a
>>> b = a
>>> b is a
>>> b is a
True
```

The state diagram looks like Figure 10.4.

> 此时的状态图如图10.4所示。

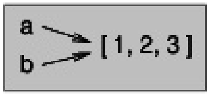


Figure 10.4: State diagram.

The association of a variable with an object is called a reference. In this example, there are two references to the same object.

> 一个变量和一个对象的关系叫做引用。在上面这个例子中，a 和 b 是对同一对象的两个引用。

An object with more than one reference has more than one name, so we say that the object is aliased.

> 这样一个对象有不止一个引用，就也有了不止一个名字，所以就说这个对象有别名了。

If the aliased object is mutable, changes made with one alias affect the other:

> 如果一个别名对象是可修改的，那么对一个别名做出的修改就会影响到其他引用：

```
>>> b[0] = 42
>>> b[0] = 42
>>> a
>>> a
[42, 2, 3]
```

Although this behavior can be useful, it is error-prone. In general, it is safer to avoid aliasing when you are working with mutable objects.

> 这一性质是很有用处的，但很容易让初学者犯错。所以一般来说，处理可变对象的时候，还是尽量避免别名使用，这样更安全些。

For immutable objects like strings, aliasing is not as much of a problem. In this example:

> 对于不可变对象比如字符串来说，别名使用就不是太大问题了。如下所示：

```
a = 'banana'
b = 'banana'
```

It almost never makes a difference whether a and b refer to the same string or not.

> a 和 b 是否指向同一个字符就基本无所谓了，几乎不会有任何影响。

10.12 List arguments 列表参数

When you pass a list to a function, the function gets a reference to the list. If the function modifies the list, the caller sees the change. For example,delete_head removes the first element from a list:

> 当你传递一个列表给一个函数的时候，函数得到的是对该列表的一个引用。如果函数修改了列表，调用者会看到变化的。比如下面这个 delete_head 函数就从列表中删除第一个元素：

```
def delete_head(t):
    del t[0]
```

Here's how it is used:

> 其用法如下：

```
>>> letters = ['a', 'b', 'c']
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> delete_head(letters)
>>> letters ['b', 'c']
>>> letters ['b', 'c']
```

The parameter t and the variable letters are aliases for the same object. The stack diagram looks like Figure 10.5.

> 形式参数 t 和变量 letters 都是同一对象的别名。栈图如图10.5所示。

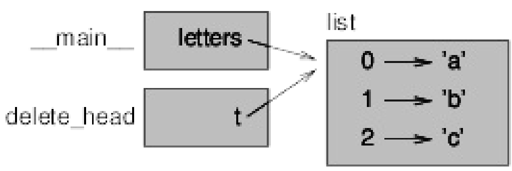


Figure 10.5: Stack diagram.

Since the list is shared by two frames, I drew it between them. It is important to distinguish between operations that modify lists and operations that create new lists. For example, the append method modifies a list, but the + operator creates a new list:

> 因为这个列表被两个框架所公用，我就把它画在了它们之间。
>
> 一定要区分修改列表的运算和产生新列表的运算，这特别重要。比如 append 方法修改一个列表，但加号+运算符是产生一个新的列表：

```
>>> t1 = [1, 2]
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> t2 = t1.append(3)
>>> t1
>>> t1
[1, 2, 3]
>>> t2
>>> t2
None
```

append modifies the list and returns None.

> append 修改了列表，返回的是空。

```
>>> t3 = t1 + [4]
>>> t3 = t1 + [4]
>>> t1
>>> t1
[1, 2, 3]
>>> t3
>>> t3
[1, 2, 3, 4]
>>> t1
>>> t1
[1, 2, 3]
```

The + operator creates a new list and leaves the original list unchanged.

> 加号+运算符创建了新的列表，并不修改源列表。

This difference is important when you write functions that are supposed to modify lists. For example, this function does not delete the head of a list:

> 这以区别相当重要，尤其是当你写一些要修改列表的函数的时候。比如下面这个函数并没有能够删除列表的第一个元素：

```
def bad_delete_head(t):
    t = t[1:]              # WRONG!
```

The slice operator creates a new list and the assignment makes t refer to it, but that doesn't affect the caller.

> 切片运算符创建了新的列表，然后赋值语句让 t 指向了这个新列表，但这不会影响调用者。

```
>>> t4 = [1, 2, 3]
>>> t4 = [1, 2, 3]
>>> bad_delete_head(t4)
>>> bad_delete_head(t4)
>>> t4
>>> t4
[1, 2, 3]
```

At the beginning of bad_delete_head, t and t4 refer to the same list. At the end,t refers to a new list, but t4 still refers to the original, unmodified list.

> 在bad_delete_head这个函数开始运行的时候，t 和 t4指向同一个列表。在结尾的时候，t 指向了新的列表，但 t4依然还是原来那个列表，而且没修改过。

An alternative is to write a function that creates and returns a new list. For example, tail returns all but the first element of a list:

> 一种替代方法是写一个能创建并返回新列表的函数。比如 tail 这个函数就返回列表除了首个元素之外的其他所有元素：

```
def tail(t):
    return t[1:]
```

This function leaves the original list unmodified. Here's how it is used:

> 这个函数会将源列表保持原样不做修改。下面是用法：

```
>>> letters = ['a', 'b', 'c']
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> rest = tail(letters)
>>> rest
>>> rest
['b', 'c']
```

10.13 Debugging 调试

Careless use of lists (and other mutable objects) can lead to long hours of debugging. Here are some common pitfalls and ways to avoid them:

> 对列表或者其他可变对象，用的不小心的货，就会带来很多麻烦，需要好长时间来调试。下面是一些常见的陷阱，以及避免的方法：

1. Most list methods modify the argument and return None. This is the opposite of the string methods, which return a new string and leave the original alone.

> 大多数列表方法都修改参数，返回空值。这正好与字符串方法相反，字符串的方法都是返回一个新字符串，保持源字符串不变。

If you are used to writing string code like this:

> 如果你习惯些下面这种字符串代码了：

```
word = word.strip()
```

It is tempting to write list code like this:

> 你估计就会写出下面这种列表代码：

```
t = t.sort()           # WRONG!
```

Because sort returns None, the next operation you perform with t is likely to fail. Before using list methods and operators, you should read the documentation carefully and then test them in interactive mode.

> 因为 sort 返回的是空值，所以对 t 的后续运算都将会失败。
>
> 在使用列表的方法和运算符之前，你应该好好读一下文档，然后在交互模式里面对它们进行一下测试。

2. Pick an idiom and stick with it. 选一种方法并坚持使用。

Part of the problem with lists is that there are too many ways to do things. For example, to remove an element from a list, you can use pop, remove,del, or even a slice assignment.

> 列表使用的问题中很大一部分都是因为有太多方法来实现目的导致的。例如要从一个列表中删除一个元素，可以用 pop，remove，del 甚至简单的切片操作。

To add an element, you can use the append method or the + operator. Assuming that t is a list and x is a list element, these are correct:

> 要加一个元素，可以用 append 方法或者加号+运算符。假设 t 是一个列表，而 x 是一个列表元素，下面的代码都是正确的：

```
t.append(x)
t = t + [x]
t += [x]
```

And these are wrong:

> 下面这就都是错的了：

```
t.append([x])          # WRONG!
t = t.append(x)        # WRONG!
t + [x]                # WRONG!
t = t + x              # WRONG!
```

Try out each of these examples in interactive mode to make sure you understand what they do. Notice that only the last one causes a runtime error; the other three are legal, but they do the wrong thing.

> 在交互模式下试试上面这些例子，确保你要理解它们的作用。要注意只有最后一个会引起运行错误，其他的三个都是合法的，但产生错误的效果。

3. Make copies to avoid aliasing. 尽量做备份，避免用别名。

If you want to use a method like sort that modifies the argument, but you need to keep the original list as well, you can make a copy.

> 如果你要用 sort 这样的方法来修改参数，又要同时保留源列表，你可以先做个备份。

```
>>> t = [3, 1, 2]
>>> t = [3, 1, 2]
>>> t2 = t[:]
>>> t2 = t[:]
>>> t2.sort()
>>> t2.sort()
>>> t
>>> t
[3, 1, 2]
>>> t2
>>> t2
[1, 2, 3]
```

In this example you could also use the built-in function sorted, which returns a new, sorted list and leaves the original alone.

> 在这个例子中，你也可以用内置函数sorted，这个函数会返回一个新的整理过的列表，而不会影响源列表。

```
>>> t2 = sorted(t)
>>> t2 = sorted(t)
>>> t
>>> t
[3, 1, 2]
>>> t2
>>> t2
[1, 2, 3]
```

10.14 Glossary 术语列表

list: A sequence of values.

> 列表：一系列值的序列。

element: One of the values in a list (or other sequence), also called items.

> 元素：一个列表或者其他序列中的值，也叫项。

nested list: A list that is an element of another list.

> 网状列表：一个作为其他列表元素的列表。

accumulator: A variable used in a loop to add up or accumulate a result.

> 累加器：一种用来在循环中累加或者拼接结果的变量。

augmented assignment: A statement that updates the value of a variable using an operator like+=.

> 增强赋值语句：使用+=这种自增运算符来更新变量值的语句。

reduce: A processing pattern that traverses a sequence and accumulates the elements into a single result.

> reduce：一种处理模式，遍历一个序列，把元素积累起来结合成一个单独的结果。

map: A processing pattern that traverses a sequence and performs an operation on each element.

> map：一种处理模式，遍历一个序列，对每一个元素都进行某种运算。

filter: A processing pattern that traverses a list and selects the elements that satisfy some criterion.

> filter：一种处理模式，遍历一个列表，选取其中满足特定规则的一些元素。

object: Something a variable can refer to. An object has a type and a value.

> 对象：变量所指向的就是对象。一个对象有特定的某个类型，以及一个值。

equivalent: Having the same value.

> 相等：有相等的值。

identical: Being the same object (which implies equivalence).

> 相同：是同一个对象（意味着必然就是相等了）。

reference: The association between a variable and its value.

> 引用：变量 a 与其值的关系。

aliasing: A circumstance where two or more variables refer to the same object.

> 别名：同一个对象有两个或者更多变量所指向的情况。

delimiter: A character or string used to indicate where a string should be split.

> 定界符：一个字符或者字符串，用来确定字符分割时候的分界。

10.15 Exercises 练习

You can download solutions to these exercises from Here.

> 你可以从 这里 下载下面这些练习的样例代码。

Exercise 1 练习1

Write a function called nested_sum that takes a list of lists of integers and adds up the elements from all of the nested lists. For example:

> 写一个函数，名为 nested_sum，接收一系列的整数列表，然后把所有分支列表中的元素加起来。如下所示：

```
>>> t = [[1, 2], [3], [4, 5, 6]]
>>> t = [[1, 2], [3], [4, 5, 6]]
>>> nested_sum(t)
>>> nested_sum(t)
21
```

Exercise 2 练习2

Write a function called cumsum that takes a list of numbers and returns the cumulative sum; that is, a new list where the ith element is the sum of the first i+1 elements from the original list. For example:

> 写一个函数，明切 cumsum，接收一个数字列表，然后返回累加的总和；也就是新列表的第 i 个元素就是源列表中前 i+1个元素的累加。如下所示：

```
>>> t = [1, 2, 3]
>>> t = [1, 2, 3]
>>> cumsum(t)
>>> cumsum(t)
[1, 3, 6]
```

Exercise 3 练习3

Write a function called middle that takes a list and returns a new list that contains all but the first and last elements. For example:

> 写一个函数，名为 middle，接收一个列表，返回一个新列表，新列表要求对源列表掐头去尾只要中间部分。如下所示：

```
>>> t = [1, 2, 3, 4]
>>> t = [1, 2, 3, 4]
>>> middle(t)
>>> middle(t)
[2, 3]
```

Exercise 4 练习4

Write a function called chop that takes a list, modifies it by removing the first and last elements, and returns None. For example:

> 写一个函数，名为 chop，接收一个列表，修改这个列表，掐头去尾，返回空值。如下所示：

```
>>> t = [1, 2, 3, 4]
>>> t = [1, 2, 3, 4]
>>> chop(t)
>>> chop(t)
>>> t
>>> t
[2, 3]
```

Exercise 5 练习5

Write a function called is_sorted that takes a list as a parameter and returns True if the list is sorted in ascending order and False otherwise. For example:

> 写一个函数，名为 is_sorted，接收一个列表作为参数，如果列表按照字母顺序升序排列，就返回真，否则返回假。如下所示：

```
>>> is_sorted([1, 2, 2])
>>> is_sorted([1, 2, 2])
True
>>> is_sorted(['b', 'a'])
>>> is_sorted(['b', 'a'])
False
```

Exercise 6 练习6

Two words are anagrams if you can rearrange the letters from one to spell the other. Write a function called is_anagram that takes two strings and returns True if they are anagrams.

> 两个单词如果可以通过顺序修改来互相转换就互为变位词。写一个函数，名为 is_anagram，接收两个字符串，如果互为变位词就返回真。

Exercise 7 练习7

Write a function called has_duplicates that takes a list and returns True if there is any element that appears more than once. It should not modify the original list.

> 写一个函数，名为 has_duplicates，接收一个列表，如果里面有重复出现的元素，就返回真。这个函数不能修改源列表。

Exercise 8 练习8

This exercise pertains to the so-called Birthday Paradox, which you can read about at Here. If there are 23 students in your class, what are the chances that two of you have the same birthday? You can estimate this probability by generating random samples of 23 birthdays and checking for matches. Hint: you can generate random birthdays with the randint function in the random module. You can download my solution from Here.

> 这个练习也可以叫做生日悖论，你可以点击这里来读一下更多背景知识。
>
> 加入你班上有23个学生，这当中两个人同一天出生的概率是多大？你可以评估一下23个随机生日中有相同生日的概率。提示一下：你可以用 randint 函数来生成随机的生日，这个函数包含在 random 模块中。
>
> 你可以从 这里下载我的样例代码。

Exercise 9 练习9

Write a function that reads the file words.txt and builds a list with one element per word. Write two versions of this function, one using the append method and the other using the idiom t = t + [x]. Which one takes longer to run? Why? Solution .

> 写一个函数，读取文件 words.txt，然后建立一个列表，这个列表中每个元素就是这个文件中的每个单词。写两个版本的这样的函数，一个使用 append 方法，另外一个用自增运算的模式：t= t + [x]。看看哪个运行时间更长？为什么会这样？
>
> 样例代码

Exercise 10 练习10

To check whether a word is in the word list, you could use the in operator, but it would be slow because it searches through the words in order.

> 要检查一个单词是不是在上面这个词汇列表里，你可以使用 in 运算符，但可能会很慢，因为这个 in 运算符要从头到尾来搜索整个词汇表。

Because the words are in alphabetical order, we can speed things up with a bisection search (also known as binary search), which is similar to what you do when you look a word up in the dictionary. You start in the middle and check to see whether the word you are looking for comes before the word in the middle of the list. If so, you search the first half of the list the same way. Otherwise you search the second half.

> 我们知道这些单词是按照字母表顺序组织的，所以我们可以加速一下，用一种对折搜索（也叫做二元搜索），这个过程就和你在现实中用字典来查单词差不多。你在中间部分开始，看看这个要搜索的词汇是不是在中间位置的前面。如果在前面，就又对前半部分取中间，继续这样来找。当然了，不在前半部分，就去后半部分找了，思路是这样的。

Either way, you cut the remaining search space in half. If the word list has 113,809 words, it will take about 17 steps to find the word or conclude that it's not there.

> 不论怎样，每次都会把搜索范围缩减到一半。如果词表包含了113809个单词，最多就是17步就能找到单词，或者能确定单词不在词汇表中。

Write a function called in_bisect that takes a sorted list and a target value and returns the index of the value in the list if it's there, or None if it's not. Or you could read the documentation of the bisect module and use that! Solution

> 那么问题来了，写一个函数，名为 in_bisect，接收一个整理过的按照字母顺序排列的列表，以及一个目标值，在列表中查找这个值，找到了就返回索引位置，找不到就返回空。
>
> 样例代码.

Exercise 11 练习11

Two words are a "reverse pair" if each is the reverse of the other. Write a program that finds all the reverse pairs in the word list. Solution.

> 两个词如果互为逆序，就称它们是『翻转配对』。写一个函数来找一下在这个词汇表中所有这样的词对。样例代码

Exercise 12 练习12

Two words "interlock" if taking alternating letters from each forms a new word. For example, "shoe" and "**cold**" interlock to form "s**c**h**o**o**l**e**d**". Solution. Credit: This exercise is inspired by an example at Here.

> 两个单词，依次拼接各自的字母，如果能组成一个新单词，就称之为『互锁』。比如，shoe 和 **cold**就可以镶嵌在一起组成组成 s**c**h**o**o**l**e**d**。（译者注：shoe+**cold**= s**c**h**o**o**l**e**d**）样例代码. 说明：这个练习受到了这里一处例子的启发。

1. Write a program that finds all pairs of words that interlock. Hint: don’t enumerate all pairs!

> 写一个程序，找到所有这样的互锁单词对。提示：不要枚举所有的单词对！

1. Can you find any words that are three-way interlocked; that is, every third letter forms a word, starting from the first, second or third?

> 你能找到那种三路互锁的单词么；就是那种三三排列三个单词的字母能组成一个单词的三个词？

Chapter 11 Dictionaries 字典

This chapter presents another built-in type called a dictionary. Dictionaries are one of Python's best features; they are the building blocks of many efficient and elegant algorithms.

> 本章要讲的内容是另外一种内置的类型，叫字典。字典是 Python 最有特色的功能之一；使用字典能构建出很多高效率又很优雅的算法。

11.1 A dictionary is a mapping 字典是一种映射

A dictionary is like a list, but more general. In a list, the indices have to be integers; in a dictionary they can be (almost) any type.

> 字典就像是一个列表一样，但更加泛化了，是列表概念的推广。在列表里面，索引必须是整数；而在字典里面，你可以用几乎任何类型来做索引了。
>
> （译者注：从字符串 string，到列表 list，再到字典 dictionary，Python 这个变量类型就是一种泛化的过程，内容在逐步推广，适用范围更大了，这里大家一定要对泛化好好理解一下，以后自己写类的时候很有用。）

A dictionary contains a collection of indices, which are called keys, and a collection of values. Each key is associated with a single value. The association of a key and a value is called a key-value pair or sometimes an item.

> 字典包括一系列的索引，不过就已经不叫索引了，而是叫键，然后还对应着一个个值，就叫键值。每个键对应着各自的一个单独的键值。这种键和键值的对应关系也叫键值对，有时候也叫项。
>
> （译者注：计算机科学上很多内容都是对数学的应用，大家真应该加油学数学啊。）

In mathematical language, a dictionary represents a mapping from keys to values, so you can also say that each key "maps to" a value. As an example, we'll build a dictionary that maps from English to Spanish words, so the keys and the values are all strings.

> 用数学语言来说，一个字典就代表了从键到键值的一种映射关系，所以你也可以说每个键映射到一个键值。举例来说，我们可以建立一个从英语单词映射到西班牙语单词的字典，这样键和简直就都是字符串了。

The function dict creates a new dictionary with no items. Because dict is the name of a built-in function, you should avoid using it as a variable name.

> dict 这个函数创建一个没有项目的空字典。因为 dict 似乎内置函数的名字了，所以你应该避免用来做变量名。

```
>>> eng2sp = dict()
>>> eng2sp = dict()
>>> eng2sp
>>> eng2sp
{}
```

The squiggly-brackets, {}, represent an empty dictionary. To add items to the dictionary, you can use square brackets:

> 大括号，也叫花括号，就是{}，代表了一个空字典。要在字典里面加项，可以使用方括号：

```
>>> eng2sp['one'] = 'uno'
>>> eng2sp['one'] = 'uno'
```

This line creates an item that maps from the key ’one’ to the value 'uno'. If we print the dictionary again, we see a key-value pair with a colon between the key and value:

> 这一行代码建立了一个项，这个项映射了键 'one' 到键值 'uno'。如果我们再来打印输出一下这个字典，就会看到里面有这样一个键值对了，键值对中间用冒号隔开了：

```
>>> eng2sp
>>> eng2sp
{'one': 'uno'}
```

This output format is also an input format. For example, you can create a new dictionary with three items:

> 这种输出的格式也可以用来输入。比如你可以这样建立一个有三个项的字典：

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

But if you print eng2sp, you might be surprised:

> 再来输出一下，你就能看到字典建好了，但顺序不一样：

```
>>> eng2sp
>>> eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

The order of the key-value pairs might not be the same. If you type the same example on your computer, you might get a different result. In general, the order of items in a dictionary is unpredictable.

> 这些键值对的顺序不一样了。如果你在你电脑上测试上面这段代码，你得到的结果也可能不一样，实际上，字典中的项的顺序是不确定的。

But that’s not a problem because the elements of a dictionary are never indexed with integer indices. Instead, you use the keys to look up the corresponding values:

> 但者其实也不要紧，因为字典里面的元素并不是用整数索引来排列的。所以你就可以直接用键来查找对应的键值：

```
>>> eng2sp['two']
>>> eng2sp['two']
'dos'
```

The key ’two’ always maps to the value 'dos' so the order of the items doesn’t matter. If the key isn’t in the dictionary, you get an exception:

> 键'two'总会映射到键值'dos'，所以项的排列顺序并不要紧。
>
> 如果你字典中没有你指定的键，你就得到如下提示：

```
>>> eng2sp['four']
>>> eng2sp['four']
KeyError: 'four'
```

The len function works on dictionaries; it returns the number of key-value pairs:

> len 函数也可以用在字典上；它会返回键值对的数目：

```
>>> len(eng2sp)
>>> len(eng2sp)
3
```

The in operator works on dictionaries, too; it tells you whether something appears as a key in the dictionary (appearing as a value is not good enough).

> in 运算符也适用于字典；你可以用它来判断某个键是不是存在于字典中（是判断键，不能判断键值）。

```
>>> 'one' in eng2sp
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
>>> 'uno' in eng2sp
False
```

To see whether something appears as a value in a dictionary, you can use the method values, which returns a collection of values, and then use the in operator:

> 要判断键值是否在字典中，你就要用到 values 方法，这个方法会把键值返回，然后用 in 判断就可以了：

```
>>> vals = eng2sp.values()
>>> vals = eng2sp.values()
>>> 'uno' in vals
>>> 'uno' in vals
True
```

The in operator uses different algorithms for lists and dictionaries. For lists, it searches the elements of the list in order, as in Section 8.6. As the list gets longer, the search time gets longer in direct proportion.

> in 运算符在字典中和列表中有不同的算法了。对列表来说，它就按照顺序搜索列表中的每一个元素，如8.6所示。随着列表越来越长了，这种搜索就消耗更多的时间，才能找到正确的位置。

For dictionaries, Python uses an algorithm called a hashtable that has a remarkable property: the in operator takes about the same amount of time no matter how many items are in the dictionary. I explain how that's possible in Section 13.4, but the explanation might not make sense until you've read a few more chapters.

> 而对字典来说，Python 使用了一种叫做哈希表的算法，这就有一种很厉害的特性：in 运算符在对字典来使用的时候无论字典规模多大，无论里面的项有多少个，花费的时间都是基本一样的。我在13.4会解释一下其实现原理，不过你要多学几章之后才能理解对此的解释。

11.2 Dictionary as a collection of counters 用字典作为计数器

Suppose you are given a string and you want to count how many times each letter appears. There are several ways you could do it:

> 假设你得到一个字符串，然后你想要查一下每个字母出现了多少次。你可以通过一下方法来实现：

1. You could create 26 variables, one for each letter of the alphabet. Then you could traverse the string and, for each character, increment the corresponding counter, probably using a chained conditional.

> 你可以建立26个变量，每一个代表一个字母。然后你遍历整个字符串，每个字母的个数都累加到对应的计数器里面，可能会用到分支条件判断。

1. You could create a list with 26 elements. Then you could convert each character to a number (using the built-in function ord), use the number as an index into the list, and increment the appropriate counter.

> 你可以建立一个有26个元素的列表。然后你把每个字母转换成一个数字（用内置的 ord 函数），用这些数字作为这个列表的索引，然后累加相应的计数器。

1. You could create a dictionary with characters as keys and counters as the corresponding values. The first time you see a character, you would add an item to the dictionary. After that you would increment the value of an existing item.

> 你可以建立一个字典，用字母作为键，用该字母出现的次数作为对应的键值。第一次遇到一个字母，就在字典里面加一个项。此后再遇到这个字母，就每次在已有的项上进行累加即可。

Each of these options performs the same computation, but each of them implements that computation in a different way.

> 上面这些方法进行的都是一样的运算，但它们各自计算的实现方法是不同的。

An implementation is a way of performing a computation; some implementations are better than others. For example, an advantage of the dictionary implementation is that we don't have to know ahead of time which letters appear in the string and we only have to make room for the letters that do appear. Here is what the code might look like:

> 实现是一种运算进行的方式；有的实现要比其他的更好一些。比如用字典来实现的优势就是我们不需要实现知道字符串中有哪些字母，只需要为其中存在的字母来提供存储空间。
>
> 下面是代码样例：

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
```

The name of the function is histogram, which is a statistical term for a collection of counters (or frequencies). The first line of the function creates an empty dictionary. The for loop traverses the string. Each time through the loop, if the character c is not in the dictionary, we create a new item with key c and the initial value 1 (since we have seen this letter once). If c is already in the dictionary we increment d[c]. Here's how it works:

> 函数的名字为 histogram，这是一个统计学术语，意思是计数（或者频次）的集合。
>
> 函数的第一行创建了一个空字典。for 循环遍历了整个字符串、每次经过循环的时候，如果字符 c 没有在字典中，就在字典中创建一个新的项，键为 c，初始值为1（因为这就算遇到一次了）。如果 c 已经存在于字典中了，就对 d[c]进行一下累加。
>
> 下面是使用的样例：

```
>>> h = histogram('brontosaurus')
>>> h = histogram('brontosaurus')
>>> h
>>> h
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

The histogram indicates that the letters ’a’ and 'b' appear once; 'o' appears twice, and so on. Dictionaries have a method called get that takes a key and a default value. If the key appears in the dictionary, get returns the corresponding value; otherwise it returns the default value. For example:

> histogram的结果表明字母a 和 b 出现了一次，o 出现了两次，等等。
>
> 字典有一个方法，叫做 get，接收一个键和一个默认值。如果这个键在字典中存在，get 就会返回对应的键值；如果不存在，它就会返回这个默认值。比如：

```
>>> h = histogram('a')
>>> h = histogram('a')
>>> h
>>> h
{'a': 1}
>>> h.get('a', 0)
>>> h.get('a', 0)
1
>>> h.get('b', 0)
>>> h.get('b', 0)
0
```

As an exercise, use get to write histogram more concisely. You should be able to eliminate the if statement.

> 做个练习，用 get 这个方法，来缩写一下 histogram 这个函数，让它更简洁些。可以去掉那些 if 语句。

11.3 Looping and dictionaries 循环与字典

If you use a dictionary in a for statement, it traverses the keys of the dictionary. For example, print_hist prints each key and the corresponding value:

> 如果你在 for 语句里面用字典，程序会遍历字典中的所有键。例如下面这个 print_hist 函数就输出其中的每一个键与对应的键值：

```
def print_hist(h):
    for c in h:
        print(c, h[c])
```

Here's what the output looks like:

> 输出如下所示：

```
>>> h = histogram('parrot')
>>> h = histogram('parrot')
>>> print_hist(h)
>>> print_hist(h)
a 1 p 1 r 2 t 1 o 1
```

Again, the keys are in no particular order. Dictionaries have a method called keys that returns the keys of the dictionary, in no particular order, as a list. As an exercise, modify print_hist to print the keys and their values in alphabetical order.

> 明显这些键的输出并没有特定顺序。字典有一个内置的叫做 keys 的方法，返回字典中的所有键成一个列表，以不确定的顺序。做个练习，修改一下上面这个 print_hist 函数，让它按照字母表的顺序输出键和键值。

11.4 Reverse lookup 逆向查找

Given a dictionary d and a key k, it is easy to find the corresponding value v = d[k]. This operation is called a lookup. But what if you have v and you want to find k? You have two problems: first, there might be more than one key that maps to the value v. Depending on the application, you might be able to pick one, or you might have to make a list that contains all of them. Second, there is no simple syntax to do a reverse lookup; you have to search. Here is a function that takes a value and returns the first key that maps to that value:

> 给定一个字典 d，以及一个键 k，很容易找到对应的键值 v=d[k]。这个操作就叫查找。
>
> 但如果你有键值 v 而要找键 k 呢？你有两个问题了：首先，可能有不止一个键的键值为 v。根据应用的不同，你也许可以从中选一个，或者就可以把所有对应的键做成一个列表。其次，没有一种简单的语法能实现这样一种逆向查找；你必须搜索一下。

```
def reverse_lookup(d, v):
    for k in d:
        if d[k] == v:
            return k
    raise LookupError()
```

This function is yet another example of the search pattern, but it uses a feature we haven't seen before, raise. The raise statement causes an exception; in this case it causes a LookupError, which is a built-in exception use to indicate that a lookup operation failed.

> 这个函数是搜索模式的另一个例子，用到了一个新的功能：raise。raise语句会导致一个异常；在这种情况下是 LookupError，这是一个内置异常，表示查找操作失败。

If we get to the end of the loop, that means v doesn't appear in the dictionary as a value, so we raise an exception. Here is an example of a successful reverse lookup:

> 如果我们运行了整个循环，就意味着 v 在字典中没有作为键值出现果，所以就 raise 一个异常回去。
>
> 下面是一个成功进行逆向查找的样例：

```
>>> h = histogram('parrot')
>>> h = histogram('parrot')
>>> k = reverse_lookup(h, 2)
>>> k = reverse_lookup(h, 2)
>>> k
>>> k
'r'
```

And an unsuccessful one:

> 下面这个是一个不成功的：

```
>>> k = reverse_lookup(h, 3)
>>> k = reverse_lookup(h, 3)
Traceback (most recent call last):   File "<stdin>", line 1, in <module>   File "<stdin>"
```

The effect when you raise an exception is the same as when Python raises one: it prints a traceback and an error message.

> 你自己 raise 一个异常的效果就和 Python 抛出的异常是一样的：程序会输出一个追溯以及一个错误信息。

The raise statement can take a detailed error message as an optional argument. For example:

> raise 语句可以给出详细的错误信息作为可选的参数。如下所示：

```
>>> raise ValueError('value does not appear in the dictionary')
>>> raise ValueError('value does not appear in the dictionary')
Traceback (most recent call last):   File "<stdin>", line 1, in ?
ValueError: value does not appear in the dictionary
```

A reverse lookup is much slower than a forward lookup; if you have to do it often, or if the dictionary gets big, the performance of your program will suffer.

> 逆向查找要比正常查找慢很多很多；如果要经常用到的话，或者字典变得很大了，程序的性能就会大打折扣。

11.5 Dictionaries and lists 字典和列表

Lists can appear as values in a dictionary. For example, if you are given a dictionary that maps from letters to frequencies, you might want to invert it; that is, create a dictionary that maps from frequencies to letters. Since there might be several letters with the same frequency, each value in the inverted dictionary should be a list of letters. Here is a function that inverts a dictionary:

> 列表可以视作字典中的值。比如给你一个字典，映射了字符与对应的频率，你可能需要逆转一下；也就是建立一个从频率映射到字母的字典。因为可能有几个字母有同样的频率，在这个逆转字典中的每个值就应该是一个字母的列表。
>
> 下面就是一个逆转字典的函数：

```
def invert_dict(d):
    inverse = dict()
    for key in d:
        val = d[key]
        if val not in inverse:
            inverse[val] = [key]
        else:
            inverse[val].append(key)
    return inverse
```

Each time through the loop, key gets a key from d and val gets the corresponding value. If val is not in inverse, that means we haven't seen it before, so we create a new item and initialize it with a singleton (a list that contains a single element). Otherwise we have seen this value before, so we append the corresponding key to the list. Here is an example:

> 每次循环的时候，key这个变量都得到 d 中的一个键，val 获取对应的键值。如果 val 不在 inverse 这个字典里面，就意味着这是首次遇到它，所以就建立一个新项，然后用一个单元素集来初始化。否则就说明这个键值已经存在了，这样我们就在对应的键的列表中添加上新的这一个键就可以了。
>
> 下面是一个样例：

```
>>> hist = histogram('parrot')
>>> hist = histogram('parrot')
>>> hist
>>> hist
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
>>> inverse = invert_dict(hist)
>>> inverse = invert_dict(hist)
>>> inverse
>>> inverse
{1: ['a', 'p', 't', 'o'], 2: ['r']}
```

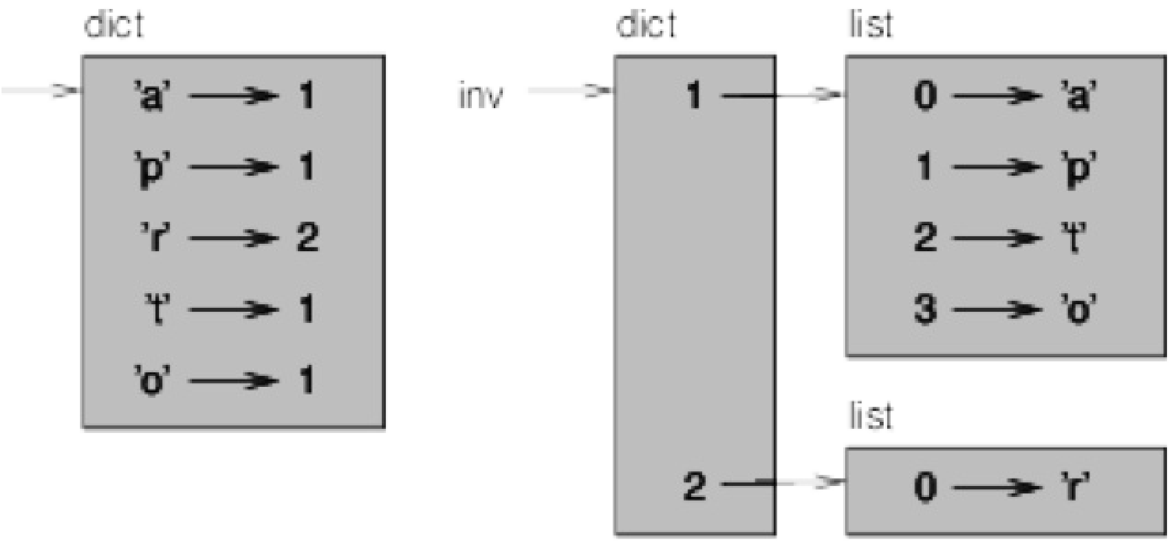


Figure 11.1: State diagram.

Figure 11.1 is a state diagram showing hist and inverse. A dictionary is represented as a box with the type dict above it and the key-value pairs inside. If the values are integers, floats or strings, I draw them inside the box, but I usually draw lists outside the box, just to keep the diagram simple. Lists can be values in a dictionary, as this example shows, but they cannot be keys. Here's what happens if you try:

> 图11.1为hist 和 inverse 两个字典的状态图。字典用方框表示，上方标示了类型 dict，方框内为键值对。如果键值为整数、浮点数或者字符串，就把它们放到一个方框内，不过通常我习惯把它们放到方框外面，这样图表看着简单干净。
>
> 如图所示，用字典中的键值组成列表，而不能用键。如果你要用键的话，就会遇到如下所示的错误：

```
>>> t = [1, 2, 3]
>>> t = [1, 2, 3]
>>> d = dict()
>>> d = dict()
>>> d[t] = 'oops'
>>> d[t] = 'oops'
Traceback (most recent call last):   File "<stdin>", line 1, in ? TypeError: list objects
```

I mentioned earlier that a dictionary is implemented using a hashtable and that means that the keys have to be hashtable.

> 我之前说过，字典是用哈希表（散列表）来实现的，这就意味着所有键都必须是散列的。

A hash is a function that takes a value (of any kind) and returns an integer. Dictionaries use these integers, called hash values, to store and look up key-value pairs.

> hash 是一个函数，接收任意一种值，然后返回一个整数。字典用这些整数来存储和查找键值对，这些整数也叫做哈希值。

This system works fine if the keys are immutable. But if the keys are mutable, like lists, bad things happen. For example, when you create a key-value pair, Python hashes the key and stores it in the corresponding location. If you modify the key and then hash it again, it would go to a different location. In that case you might have two entries for the same key, or you might not be able to find a key. Either way, the dictionary wouldn't work correctly.

> 如果键不可修改，系统工作正常。但如果键可以修改，比如是列表，就悲剧了。例如，你创建一个键值对的时候，Python 计算键的哈希值，然后存在相应的位置。如果你修改了键，然后在计算哈希值，就不会指向同一个位置了。这时候一个键就可以有两个指向了，或者你就可能找不到某个键了。总之字典都不能正常工作了。

That's why keys have to be hashable, and why mutable types like lists aren't. The simplest way to get around this limitation is to use tuples, which we will see in the next chapter. Since dictionaries are mutable, they can't be used as keys, but they can be used as values.

> 这就是为什么这些键必须是散列的，而像列表这样的可变类型就不行。解决这个问题的最简单方式就是使用元组，这个我们会在下一章来学习。
>
> 因为字典是可以修改的，所以不能用来做键，只能用来做键值。
>
> （译者注：哈希表是一种散列表，相关内容译者知道的太少，所以这段翻译的质量大打折扣，实在抱歉。）

11.6 Memos 备忘

If you played with the fibonacci function from Section 6.7, you might have noticed that the bigger the argument you provide, the longer the function takes to run. Furthermore, the run time increases quickly. To understand why, consider Figure 11.2, which shows the call graph forfibonacci with n=4:

> 如果你试过了6.7中提到的斐波那契数列，你估计会发现随着参数增大，函数运行的时间也变长了。另外，运行时间的增长很显著。
>
> 要理解这是怎么回事，就要参考一下图11.2，图中展示了当 n=4的时候函数调用的情况。

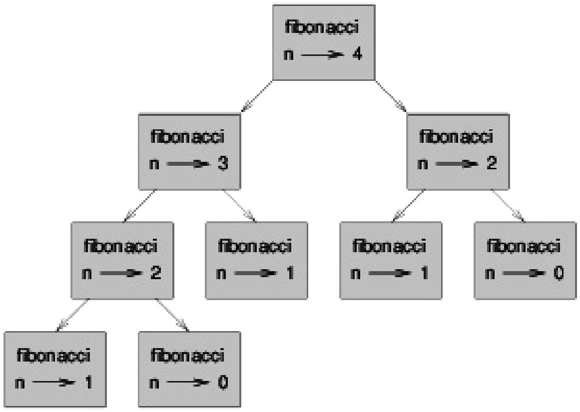


Figure 11.2: Call graph.

A call graph shows a set of function frames, with lines connecting each frame to the frames of the functions it calls. At the top of the graph, fibonacci with n=4 calls fibonacci with n=3 and n=2. In turn, fibonacci with n=3 calls fibonacciwith n=2 and n=1. And so on.

> 调用图展示了一系列的函数图框，图框直接的连线表示了函数只见的调用关系。顶层位置函数的参数 n =4，调用了 n=3和 n=2两种情况的函数。相应的 n=3的时候要调用 n=2和 n=1两种情况。依此类推。

Count how many times fibonacci(0) and fibonacci(1) are called. This is an inefficient solution to the problem, and it gets worse as the argument gets bigger.

> 算算fibonacci(0)和fibonacci(1)要被调用多少次吧。这样的解决方案是低效率的，随着参数增大，效率就越来越低了。

One solution is to keep track of values that have already been computed by storing them in a dictionary. A previously computed value that is stored for later use is called a memo. Here is a “memoized” version of fibonacci:

> 另外一种思路就是保存一下已经被计算过的值，然后保存在一个字典中。之前计算过的值存储起来，这样后续的运算中能够使用，这就叫备忘。下面是一个用这种思路来实现的斐波那契函数：

```
known = {0:0, 1:1}
def fibonacci(n):
    if n in known:
        return known[n]
    res = fibonacci(n-1) + fibonacci(n-2)
    known[n] = res
    return res
```

known is a dictionary that keeps track of the Fibonacci numbers we already know. It starts with two items: 0 maps to 0 and 1 maps to 1.

> known 是一个用来保存已经计算斐波那契函数值的字典。开始项目有两个，0对应0，1对应1，各自分别是各自的斐波那契函数值。

Whenever fibonacci is called, it checks known. If the result is already there, it can return immediately. Otherwise it has to compute the new value, add it to the dictionary, and return it. If you run this version of fibonacci and compare it with the original, you will find that it is much faster.

> 这样只要斐波那契函数被调用了，就会检查 known 这个字典，如果里面有计算过的可用结果，就立即返回。不然的话就计算出新的值，并且存到字典里面，然后返回这个新计算的值。
>
> 如果你运行这一个版本的斐波那契函数，你会发现比原来那个版本要快得多。

11.7 Global variables 全局变量

In the previous example, known is created outside the function, so it belongs to the special frame called __main__. Variables in __main__ are sometimes called global because they can be accessed from any function. Unlike local variables, which disappear when their function ends, global variables persist from one function call to the next.

> 在上面的例子中，known 这个字典是在函数外创建的，所以它属于主函数内，这是一个特殊的层。在主函数中的变量也叫全局变量，因为所有函数都可以访问这些变量。局部变量在所属的函数结束后就消失了，而主函数在其他函数调用结束后依然还存在。

It is common to use global variables for flags; that is, boolean variables that indicate (“flag”) whether a condition is true. For example, some programs use a flag named verbose to control the level of detail in the output:

> 一般常用全局变量作为 flag，也就是标识；比如用来判断一个条件是否成立的布尔变量之类的。比如有的程序用名字为 verbose 的标识变量，来控制输出内容的详细程度：

```
verbose = True
def example1():
    if verbose:
        print('Running example1')
```

If you try to reassign a global variable, you might be surprised. The following example is supposed to keep track of whether the function has been called:

> 如果你想给全局变量重新赋值，结果会很意外。下面的例子中，本来是想要追踪确定函数是否被调用了：

```
been_called = False
def example2():
    been_called = True         # WRONG
```

But if you run it you will see that the value of been_called doesn't change. The problem is that example2 creates a new local variable named been_called. The local variable goes away when the function ends, and has no effect on the global variable.

> 你可以运行一下，并不报错，只是 been_called 的值并不会变化。这个情况的原因是 example2这个函数创建了一个新的名为 been_called 的局部变量。函数结束之后，局部变量就释放了，并不会影响全局变量。

To reassign a global variable inside a function you have to declare the global variable before you use it:

> 要在函数内部来给全局变量重新赋值，必须要在使用之前声明这个全局变量：

```
been_called = False
    def example2():
        global been_called
        been_called = True
```

The global statement tells the interpreter something like, "In this function, when I say been_called, I mean the global variable; don't create a local one." Here's an example that tries to update a global variable:

> global 那句代码的效果是告诉解释器：『在这个函数内，been_called 使之全局变量；不要创建一个同名的局部变量。』
>
> 下面的例子中，试图对全局变量进行更新：

```
count = 0
def example3():
    count = count + 1          # WRONG
```

If you run it you get:

> 运行的话，你会得到如下提示：

```
UnboundLocalError: local variable 'count' referenced before assignment
```

> 译者注：错误提示的意思是未绑定局部错误：局部变量 count 未经赋值就被引用。

Python assumes that count is local, and under that assumption you are reading it before writing it. The solution, again, is to declare count global.

> Python 会假设这个 count 是局部的，然后基于这样的假设，你就是在写出该变量之前就试图读取。这样问题的解决方法依然就是声称count 为全局变量。

```
def example3():
    global count
    count += 1
```

If a global variable refers to a mutable value, you can modify the value without declaring the variable:

> 如果全局变量指向的是一个可修改的值，你可以无需声明该变量就直接修改：

```
known = {0:0, 1:1}
def example4():
    known[2] = 1
```

So you can add, remove and replace elements of a global list or dictionary, but if you want to reassign the variable, you have to declare it:

> 所以你可以在全局的列表或者字典里面添加、删除或者替换元素，但如果你要重新给这个全局变量赋值，就必须要声明了：

```
def example5():
    global known
    known = dict()
```

Global variables can be useful, but if you have a lot of them, and you modify them frequently, they can make programs hard to debug.

> 全局变量很有用，但不能滥用，要是总修改全局变量的值，就让程序很难调试了。

11.8 Debugging 调试

As you work with bigger datasets it can become unwieldy to debug by printing and checking the output by hand. Here are some suggestions for debugging large datasets:

> 现在数据结构逐渐复杂了，再用打印输出和手动检验的方法来调试就很费劲了。下面是一些对这种复杂数据结构下的建议：

Scale down the input: If possible, reduce the size of the dataset. For example if the program reads a text file, start with just the first 10 lines, or with the smallest example you can find. You can either edit the files themselves, or (better) modify the program so it reads only the first n lines.

> 缩减输入：
>
> 尽可能缩小数据的规模。如果程序要读取一个文本文档，而只读前面的十行，或者用你能找到的最小规模的样例。你可以编辑一下文件本身，或者直接修改程序来仅读取前面的 n 行，这样更好。

If there is an error, you can reduce n to the smallest value that manifests the error, and then increase it gradually as you find and correct errors.

> 如果存在错误了，你可以减小一下 n，一直到错误存在的最小的 n 值，然后再逐渐增加 n，这样就能找到错误并改正了。

Check summaries and types: Instead of printing and checking the entire dataset, consider printing summaries of the data: for example, the number of items in a dictionary or the total of a list of numbers.

> 检查概要和类型：
>
> 这回咱就不再打印检查整个数据表，而是打印输出数据的概要：比如字典中的项的个数，或者一个列表中的数目总和。

A common cause of runtime errors is a value that is not the right type. For debugging this kind of error, it is often enough to print the type of a value.

> 导致运行错误的一种常见原因就是类型错误。对这类错误进行调试，输出一下值的类型就可以了。

Write self-checks: Sometimes you can write code to check for errors automatically. For example, if you are computing the average of a list of numbers, you could check that the result is not greater than the largest element in the list or less than the smallest. This is called a “sanity check” because it detects results that are “insane”. Another kind of check compares the results of two different computations to see if they are consistent. This is called a “consistency check”.

> 写自检代码：
>
> 有时你也可以写自动检查错误的代码。举例来说，假如你计算一个列表中数字的平均值，你可以检查一下结果是不是比列表中的最大值还大或者比最小值还小。这也叫『心智检查』，因为是来检查结果是否『疯了』（译者注：也就是错得很荒诞的意思。）另外一种检查方法是用两种不同运算，然后对比结果，看看他们是否一致。后面这种叫『一致性检查』。

Format the output: Formatting debugging output can make it easier to spot an error. We saw an example in Section 6.9. The pprint module provides a pprint function that displays built-in types in a more human-readable format (pprint stands for “pretty print”).

> 格式化输出：
>
> 格式化的调试输出，更容易找到错误。在6.9的时候我们见过一个例子了。pprint 模块内置了一个 pprint 函数，该函数能够把内置的类型用人读起来更容易的格式来显示出来（pprint 就是『pretty print』的缩写）。

Again, time you spend building scaffolding can reduce the time you spend debugging.

> 再次强调一下，搭建脚手架代码的时间越长，用来调试的时间就会相应地缩短。

11.9 Glossary 术语列表

mapping: A relationship in which each element of one set corresponds to an element of another set.

> 映射：一组数据中元素与另一组数据中元素的一一对应的关系。

dictionary: A mapping from keys to their corresponding values.

> 字典：从键到对应键值的映射。

key-value pair: The representation of the mapping from a key to a value.

> 键值对：有映射关系的一对键和对应的键值。

item: In a dictionary, another name for a key-value pair.

> 项：字典中键值对也叫项。

key: An object that appears in a dictionary as the first part of a key-value pair.

> 键：字典中的一个对象，键值对中的第一部分。

value: An object that appears in a dictionary as the second part of a key-value pair. This is more specific than our previous use of the word “value”.

> 键值：字典中的一个对象，键值对的第二部分。这个和之前提到的值不同，在字典使用过程中指代的是键值，而不是数值。

implementation: A way of performing a computation.

> 实现：进行计算的一种方式。

hashtable: The algorithm used to implement Python dictionaries.

> 哈希表：Python 实现字典的一种算法。

hash function: A function used by a hashtable to compute the location for a key.

> 哈希函数：哈希表使用的一种函数，能计算出一个键的位置。

hashable: A type that has a hash function. Immutable types like integers, floats and strings are hashable; mutable types like lists and dictionaries are not.

> 散列的：一种类型，有哈希函数。不可变类型比如整形、浮点数和字符串都是散列的；可变类型比如列表和字典则不是。
>
> （译者注：这段我翻译的很狗，因为术语不是很熟悉，等有空我再查查去。）

lookup: A dictionary operation that takes a key and finds the corresponding value.

> 查找：字典操作的一种，根据已有的键查找对应的键值。

reverse lookup: A dictionary operation that takes a value and finds one or more keys that map to it.

> 逆向查找：字典操作的一种，根据一个键值找对应的一个或者多个键。

raise statement: A statement that (deliberately) raises an exception.

> raise 语句：特地要求抛出异常的一个语句。

singleton: A list (or other sequence) with a single element.

> 单元素集：只含有一个单独元素的列表或者其他序列。

call graph: A diagram that shows every frame created during the execution of a program, with an arrow from each caller to each callee.

> 调用图：一种图解，解释程序运行过程中每一个步骤，用箭头来来连接调用者和被调用者之间。

memo: A computed value stored to avoid unnecessary future computation.

> 备忘：将计算得到的值存储起来，避免后续的额外计算。

global variable: A variable defined outside a function. Global variables can be accessed from any function.

> 全局变量：函数外定义的变量。全局变量能被所有函数来读取使用。

global statement: A statement that declares a variable name global.

> global 语句：声明一个变量为全局的语句。

flag: A boolean variable used to indicate whether a condition is true.

> 标识：一个布尔变量，用来指示一个条件是否为真。

declaration: A statement like global that tells the interpreter something about a variable.

> 声明：比如 global 这样的语句，用来告诉解释器变量的特征。

11.10 Exercises 练习

Exercise 1 练习1

Write a function that reads the words in words.txt and stores them as keys in a dictionary. It doesn't matter what the values are. Then you can use the in operator as a fast way to check whether a string is in the dictionary. If you did Exercise 10, you can compare the speed of this implementation with the list in operator and the bisection search.

> 写一个函数来读取 words.txt 文件中的单词，然后作为键存到一个字典中。键值是什么不要紧。然后用 in 运算符来快速检查一个字符串是否在字典中。
>
> 如果你做过第十章的练习，你可以对比一下这种实现和列表中的 in 运算符以及对折搜索的速度。

Exercise 2 练习2

Read the documentation of the dictionary method setdefault and use it to write a more concise version of invert_dict. (Solution)[http://thinkpython2.com/code/invert_dict.py].

> 读一下字典中 setdefault 方法的相关文档，然后用这个方法来写一个更精简版本的 invert_dict 函数。 (S样例代码)[http://thinkpython2.com/code/invert_dict.py]。

Exercise 3 练习3

Memoize the Ackermann function from Exercise 2 and see if memoization makes it possible to evaluate the function with bigger arguments. Hint: no. (Solution) [http://thinkpython2.com/code/ackermann_memo.py].

> 用备忘的方法来改进一下第二章练习中的Ackermann函数，看看是不是能让让函数处理更大的参数。提示：不行。(样例代码)
> [http://thinkpython2.com/code/ackermann_memo.py]。

Exercise 4 练习4

If you did Exercise 7, you already have a function named has_duplicates that takes a list as a parameter and returns True if there is any object that appears more than once in the list. Use a dictionary to write a faster, simpler version of has_duplicates. (Solution) [http://thinkpython2.com/code/has_duplicates.py].

> 如果你做过了第七章的练习，应该已经写过一个名叫 has_duplicates 的函数了，这个函数用列表做参数，如果里面有元素出现了重复，就返回真。
>
> 用字典来写一个更快速更简单的版本。(样例代码)[http://thinkpython2.com/code/has_duplicates.py]。

Exercise 5 练习5

Two words are “rotate pairs” if you can rotate one of them and get the other (see rotate_word in Exercise 5). Write a program that reads a word list and finds all the rotate pairs. (Solution)[http://thinkpython2.com/code/rotate_pairs.py].

> 一个词如果翻转顺序成为另外一个词，这两个词就为『翻转词对』（参见第五章练习的rotate_word，译者注：作者这个练习我没找到。。。）。
>
> 写一个函数读取一个单词表，然后找到所有这样的单词对。(样例代码)[http://thinkpython2.com/code/rotate_pairs.py].

Exercise 6 练习6

Here’s another Puzzler from (Car Talk)[http://www.cartalk.com/content/puzzlers]: This was sent in by a fellow named Dan O’Leary. He came upon a common one-syllable, five-letter word recently that has the following unique property. When you remove the first letter, the remaining letters form a homophone of the original word, that is a word that sounds exactly the same. Replace the first letter, that is, put it back and remove the second letter and the result is yet another homophone of the original word. And the question is, what’s the word?

> 下面是一个来自(Car Talk)[http://www.cartalk.com/content/puzzlers]的谜语：
>
> 这条谜语来自一个名叫 Dan O'Leary的朋友。他最近发现一个单词，这个单词有一个音节，五个字母，然后有以下所述的特定性质。
>
> 去掉第一个字母，得到的是与原词同音异形异义词，发音与原词一模一样。替换一下首字母，也就是把第一个字母放回去，然后把第二个字母去掉，得到的是另外一个这样的同音异形异义词。那么问题来了，这是个什么词呢？

Now I’m going to give you an example that doesn’t work. Let’s look at the five-letter word, ‘wrack.’ W-R-A-C-K, you know like to ‘wrack with pain.’ If I remove the first letter, I am left with a four-letter word, ’R-A-C-K.’ As in, ‘Holy cow, did you see the rack on that buck! It must have been a nine-pointer!’ It’s a perfect homophone. If you put the ‘w’ back, and remove the ‘r,’ instead, you’re left with the word, ‘wack,’ which is a real word, it’s just not a homophone of the other two words.

> 现在我给你提供一个错误的例子。咱们先看一下五个字母的单词，「wrack」。去掉第一个字母，得到的四个字母单词是「R-A-C-K」。但去掉第二个字母得到的是「W-A-C-K」，这就不是前两个词的同音异形异义词。（译者注：词义的细节就略去了，没有太大必要。）

But there is, however, at least one word that Dan and we know of, which will yield two homophones if you remove either of the first two letters to make two, new four-letter words. The question is, what's the word?

> 但这个词至少有一个，Dan 和咱们都知道的，分别删除前两个字母会产生两个同音异形异义的四个字母的单词。问题就是，这是哪个词？

You can use the dictionary from Exercise 1 to check whether a string is in the word list. To check whether two words are homophones, you can use the CMU Pronouncing Dictionary. You can download it from (Here)[http://www.speech.cs.cmu.edu/cgi-bin/cmudict] or from (Here)[http://thinkpython2.com/code/c06d] and you can also download (Here)[http://thinkpython2.com/code/pronounce.py], which provides a function namedread_dictionary that reads the pronouncing dictionary and returns a Python dictionary that maps from each word to a string that describes its primary pronunciation. Write a program that lists all the words that solve the Puzzler. (Solution)[http://thinkpython2.com/code/homophone.py].

> 你可以用本章练习1的字典来检查一个字符串是否在一个字典之中。检查两个单词是不是同音异形异义词，可以用 CMU 发音字典。可以从(这里)[http://www.speech.cs.cmu.edu/cgi-bin/cmudict]或者(这里)[http://thinkpython2.com/code/c06d]或者(这里)[http://thinkpython2.com/code/pronounce.py]来下载，该字典提供了一个名为 read_dictionary的函数，该函数会读取发音词典，然后返回一个 Python 词典，返回的这个词典会映射每一个单词到描述单词读音的字符串。

写一个函数来找到所有满足谜语要求的单词。(样例代码)[http://thinkpython2.com/code/homophone.py]。

Chapter 12 Tuples 元组

This chapter presents one more built-in type, the tuple, and then shows how lists, dictionaries, and tuples work together. I also present a useful feature for variable-length argument lists, the gather and scatter operators.

> 本章我们要说的是另外一种内置类型，元组，以及列表、字典和元组如何协同工作。此外还有一个非常有用的功能：可变长度的列表，聚集和分散运算符。

One note: there is no consensus on how to pronounce “tuple”. Some people say “tuh-ple”, which rhymes with “supple”. But in the context of programming, most people say “too-ple”, which rhymes with “quadruple”.

> 一点提示：元组的英文单词 tuple 怎么读还有争议。有人认为是发[tʌpəl] 的音，就跟『supple』里面的一样读音。但编程语境下，大家普遍读[tu:pəl]，跟『quadruple』里一样。

12.1 Tuples are immutable 元组不可修改

A tuple is a sequence of values. The values can be any type, and they are indexed by integers, so in that respect tuples are a lot like lists. The important difference is that tuples are immutable. Syntactically, a tuple is a comma-separated list of values:

> 元组是一系列的值。这些值可以是任意类型的，并且用整数序号作为索引，所以可以发现元组和列表非常相似。二者间重要的区别就是元组是不可修改的。
>
> 元组的语法是一系列用逗号分隔的值：

```
>>> t = 'a', 'b', 'c', 'd', 'e'
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Although it is not necessary, it is common to enclose tuples in parentheses:

> 通常都用一对圆括号把元组的元素包括起来，当然不这样也没事。

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t = ('a', 'b', 'c', 'd', 'e')
```

To create a tuple with a single element, you have to include a final comma:

> 要建立一个单个元素构成的元组，必须要在结尾加上逗号：

```
>>> t1 = 'a',
>>> t1 = 'a',
>>> type(t1)
>>> type(t1)
<class 'tuple'>
```

A value in parentheses is not a tuple:

> 只用括号放一个值则并不是元组：

```
>>> t2 = ('a')
>>> t2 = ('a')
>>> type(t2)
>>> type(t2)
<class 'str'>
```

Another way to create a tuple is the built-in function tuple. With no argument, it creates an empty tuple:

> 另一中建立元组的方法是使用内置函数 tuple。不提供参数的情况下，默认就建立一个空的元组。

```
>>> t = tuple()
>>> t = tuple()
>>> t
>>> t
()
```

If the argument is a sequence (string, list or tuple), the result is a tuple with the elements of the sequence:

> 如果参数为一个序列（比如字符串、列表或者元组），结果就会得到一个以该序列元素组成的元组。

```
>>> t = tuple('lupins')
>>> t = tuple('lupins')
>>> t
>>> t
('l', 'u', 'p', 'i', 'n', 's')
```

Because tuple is the name of a built-in function, you should avoid using it as a variable name. Most list operators also work on tuples. The bracket operator indexes an element:

> tuple 是内置函数命了，所以你就不能用来作为变量名了。
>
> 列表的各种运算符也基本适用于元组。方括号可以用来索引元素：

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t[0]
>>> t[0]
'a'
```

And the slice operator selects a range of elements.

> 切片运算符也可以用于选取某一区间的元素。

```
>>> t[1:3]
>>> t[1:3]
('b', 'c')
```

But if you try to modify one of the elements of the tuple, you get an error:

> 但如果你想修改元组中的某个元素，就会得到错误了：

```
>>> t[0] = 'A'
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

Because tuples are immutable, you can't modify the elements. But you can replace one tuple with another:

> 因为元组是不能修改的，你不能修改其中的元素。但是可以用另一个元组来替换已有的元组。

```
>>> t = ('A',) + t[1:]
>>> t = ('A',) + t[1:]
>>> t
>>> t
('A', 'b', 'c', 'd', 'e')
```

This statement makes a new tuple and then makes t refer to it. The relational operators work with tuples and other sequences; Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next elements, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big).

> 上面这个语句建立了一个新的元组，然后让 t 指向了这个新的元组。
>
> 关系运算符也适用于元组和其他序列；Python 从每个元素的首个元素开始对比。如果相等，就对比下一个元素，依此类推，之道找到不同元素为止。
>
> 有了不同元素之后，后面的其他元素就被忽略掉了（即便很大也没用）。

```
>>> (0, 1, 2) < (0, 3, 4)
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

12.2 Tuple assignment 元组赋值

It is often useful to swap the values of two variables. With conventional assignments, you have to use a temporary variable. For example, to swap a and b:

> 对两个变量的值进行交换是一种常用操作。用常见语句来实现的话，就必须有一个临时变量。比如下面这个例子中是交换 a 和 b：

```
>>> temp = a
>>> temp = a
>>> a = b
>>> a = b
>>> b = temp
>>> b = temp
```

This solution is cumbersome; tuple assignment is more elegant:

> 这样解决还是挺麻烦的；用元组赋值就更简洁了：

```
>>> a, b = b, a
>>> a, b = b, a
```

The left side is a tuple of variables; the right side is a tuple of expressions. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments. The number of variables on the left and the number of values on the right have to be the same:

> 等号左边的是变量组成的一个元组；右边的是表达式的元组。每个值都被赋给了对应的变量。等号右边的表达式的值保留了赋值之前的初始值。
>
> 等号左右两侧的变量和值的数目都必须是一样的。

```
>>> a, b = 1, 2, 3
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

More generally, the right side can be any kind of sequence (string, list or tuple). For example, to split an email address into a user name and a domain, you could write:

> 更普适的情况下，等号右边以是任意一种序列（字符串、列表或者元组）。比如，要把一个电子邮件地址转换成一个用户名和一个域名，可以用如下代码实现：

```
>>> addr = 'monty@python.org'
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
>>> uname, domain = addr.split('@')
```

The return value from split is a list with two elements; the first element is assigned to uname, the second to domain.

> split 的返回值是一个有两个元素的列表；第一个元素赋值给了 uname 这个变量，第二个赋值给了 domain 这个变量。

```
>>> uname
>>> uname
'monty'
>>> domain
>>> domain
'python.org'
```

12.3 Tuples as return values 用元组做返回值

Strictly speaking, a function can only return one value, but if the value is a tuple, the effect is the same as returning multiple values. For example, if you want to divide two integers and compute the quotient and remainder, it is inefficient to compute x/y and then x%y. It is better to compute them both at the same time.

> 严格来说，一个函数只能返回一个值，但如果这个值是一个元组，效果就和返回多个值一样了。例如，如果你想要将两个整数相除，计算商和余数，如果要分开计算 x/y 以及 x%y 就很麻烦了。更好的办法是同时计算这两个值。

The built-in function divmod takes two arguments and returns a tuple of two values, the quotient and remainder. You can store the result as a tuple:

> 内置函数 divmod 就会接收两个参数，然后返回一个有两个值的元组，这两个值分别为商和余数。
>
> 可以把结果存储为一个元组：

```
>>> t = divmod(7, 3)
>>> t = divmod(7, 3)
>>> t
>>> t
(2, 1)
```

Or use tuple assignment to store the elements separately:

> 或者可以用元组赋值来分别存储这两个值：

```
>>> quot, rem = divmod(7, 3)
>>> quot, rem = divmod(7, 3)
>>> quot
>>> quot
2
>>> rem
>>> rem
1
```

Here is an example of a function that returns a tuple:

> 下面的例子中，函数返回一个元组作为返回值：

```
def min_max(t):
    return min(t), max(t)
```

max and min are built-in functions that find the largest and smallest elements of a sequence. min_max computes both and returns a tuple of two values.

> max 和 min 都是内置函数，会找到序列中的最大值或者最小值，min_max 这个函数会同时求得最大值和最小值，然后把这两个值作为元组来返回。

12.4 Variable-length argument tuples 参数长度可变的元组

Functions can take a variable number of arguments. A parameter name that begins with * gathers arguments into a tuple. For example, printall takes any number of arguments and prints them:

> 函数的参数可以有任意多个。用星号*开头来作为形式参数名，可以将所有实际参数收录到一个元组中。例如 printall 就可以获取任意多个数的参数，然后把它们都打印输出：

```
def printall(*args):
    print(args)
```

The gather parameter can have any name you like, but args is conventional. Here's how the function works:

> 你可以随意命名收集来的这些参数，但 args 这个是约定俗成的惯例。下面展示一下这个函数如何使用：

```
>>> printall(1, 2.0, '3')
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

The complement of gather is scatter. If you have a sequence of values and you want to pass it to a function as multiple arguments, you can use the *operator. For example, divmod takes exactly two arguments; it doesn't work with a tuple:

> 与聚集相对的就是分散了。如果有一系列的值，然后想把它们作为多个参数传递给一个函数，就可以用星号*运算符。比如 divmod 要求必须是两个参数；如果给它一个元组，是不能进行运算的：

```
>>> t = (7, 3)
>>> t = (7, 3)
>>> divmod(t)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
```

But if you scatter the tuple, it works:

> 但如果拆分这个元组，就可以了：

```
>>> divmod(*t)
>>> divmod(*t)
(2, 1)
```

Many of the built-in functions use variable-length argument tuples. For example, max and min can take any number of arguments:

> 很多内置函数都用到了参数长度可变的元组。比如 max 和 min 就可以接收任意数量的参数：

```
>>> max(1, 2, 3)
>>> max(1, 2, 3)
3
```

But sum does not.

> 但求和函数 sum 就不行了。

```
>>> sum(1, 2, 3)
>>> sum(1, 2, 3)
TypeError: sum expected at most 2 arguments, got 3
```

As an exercise, write a function called sumall that takes any number of arguments and returns their sum.

> 做个练习，写一个名为 sumall 的函数，让它可以接收任意数量的参数，返回总和。

12.5 Lists and tuples 列表和元组

zip is a built-in function that takes two or more sequences and returns a list of tuples where each tuple contains one element from each sequence. The name of the function refers to a zipper, which joins and interleaves two rows of teeth. This example zips a string and a list:

> zip 是一个内置函数，接收两个或更多的序列作为参数，然后返回返回一个元组列表，该列表中每个元组都包含了从各个序列中的一个元素。这个函数名的意思就是拉锁，就是把不相关的两排拉锁齿连接到一起。
>
> 下面这个例子中，一个字符串和一个列表通过 zip 这个函数连接到了一起：

```
>>> s = 'abc'
>>> s = 'abc'
>>> t = [0, 1, 2]
>>> t = [0, 1, 2]
>>> zip(s, t)
>>> zip(s, t)
<zip object at 0x7f7d0a9e7c48>
```

The result is a zip object that knows how to iterate through the pairs. The most common use of zip is in a for loop:

> 该函数的返回值是一个 zip 对象，该对象可以用来迭代所有的数值对。zip 函数经常被用到 for 循环中：

```
>>> for pair in zip(s, t): ...
>>> for pair in zip(s, t): ...
print(pair) ...
('a', 0) ('b', 1) ('c', 2)
```

A zip object is a kind of iterator, which is any object that iterates through a sequence. Iterators are similar to lists in some ways, but unlike lists, you can't use an index to select an element from an iterator. If you want to use list operators and methods, you can use a zip object to make a list:

> zip 对象是一种迭代器，也就是某种可以迭代整个序列的对象。迭代器和列表有些相似，但不同于列表的是，你无法通过索引来选择迭代器中的指定元素。
>
> 如果想用列表的运算符和方法，可以用 zip 对象来构成一个列表：

```
>>> list(zip(s, t))
>>> list(zip(s, t))
[('a', 0), ('b', 1), ('c', 2)]
```

The result is a list of tuples; in this example, each tuple contains a character from the string and the corresponding element from the list. If the sequences are not the same length, the result has the length of the shorter one.

> 返回值是一个由元组构成的列表；在这个例子中，每个元组都包含了字符串中的一个字母，以及列表中对应位置的元素。
>
> 在长度不同的序列中，返回的结果长度取决于最短的一个。

```
>>> list(zip('Anne', 'Elk'))
>>> list(zip('Anne', 'Elk'))
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

You can use tuple assignment in a for loop to traverse a list of tuples:

> 用 for 循环来遍历一个元组列表的时候，可以用元组赋值语句：

```
t = [('a', 0), ('b', 1), ('c', 2)]
for letter, number in t:
    print(number, letter)
```

Each time through the loop, Python selects the next tuple in the list and assigns the elements to letter and number. The output of this loop is:

> 每次经历循环的时候，Python 都选中列表中的下一个元组，然后把元素赋值给字母和数字。该循环的输出如下：

```
0 a 1 b 2 c
```

If you combine zip, for and tuple assignment, you get a useful idiom for traversing two (or more) sequences at the same time. For example, has_match takes two sequences, t1 and t2, and returns True if there is an index i such that t1[i] == t2[i]:

> 如果结合使用 zip、for 循环以及元组赋值，就能得到一种能同时遍历两个以上序列的代码组合。比如下面例子中的 has_match 这个函数，接收两个序列t1和 t2作为参数，然后如果存在一个索引位置 i 使得 t1[i] == t2[i]就返回真：

```
def has_match(t1, t2):
    for x, y in zip(t1, t2):
        if x == y:
            return True
    return False
```

If you need to traverse the elements of a sequence and their indices, you can use the built-in function enumerate:

> 如果你要遍历一个序列中的所有元素以及它们的索引，可以用内置的函数 enumerate：

```
for index, element in enumerate('abc'):
    print(index, element)
```

The result from enumerate is an enumerate object, which iterates a sequence of pairs; each pair contains an index (starting from 0) and an element from the given sequence. In this example, the output is

```
0 a 1 b 2 c
```

Again.

> enumerate 函数的返回值是一个枚举对象，它会遍历整个成对序列；每一对都包括一个索引（从0开始）以及给定序列的一个元素。在本节的例子中，输出依然如下：
>
> ```
> 0 a 1 b 2 c
> ```
>
> ## 12.6 Dictionaries and tuples 词典与元组

Dictionaries have a method called items that returns a sequence of tuples, where each tuple is a key-value pair.

> 字典有一个名为 items 的方法，会返回一个由元组组成的序列，每一个元组都是字典中的一个键值对。

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> t = d.items()
>>> t
>>> t
dict_items([('c', 2), ('a', 0), ('b', 1)])
```

The result is a dict_items object, which is an iterator that iterates the key-value pairs. You can use it in a for loop like this:

> 结果是一个 dict_items 对象，这是一个迭代器，迭代所有的键值对。可以在 for 循环里面用这个对象，如下所示：

```
>>> for key, value in d.items():
>>> for key, value in d.items():
...     print(key, value)
... c 2 a 0 b 1
```

As you should expect from a dictionary, the items are in no particular order. Going in the other direction, you can use a list of tuples to initialize a new dictionary:

> 你也应该预料到了，字典里面的项是没有固定顺序的。
>
> 反过来使用的话，你就也可以用一个元组的列表来初始化一个新的字典：

```
>>> t = [('a', 0), ('c', 2), ('b', 1)]
>>> t = [('a', 0), ('c', 2), ('b', 1)]
>>> d = dict(t)
>>> d = dict(t)
>>> d
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

Combining dict with zip yields a concise way to create a dictionary:

> 结合使用 dict 和 zip， 会得到一种建立字典的简便方法：

```
>>> d = dict(zip('abc', range(3)))
>>> d = dict(zip('abc', range(3)))
>>> d
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

The dictionary method update also takes a list of tuples and adds them, as key-value pairs, to an existing dictionary.

> 字典的 update 方法也接收一个元组列表，然后把它们作为键值对添加到一个已存在的字典中。

It is common to use tuples as keys in dictionaries (primarily because you can't use lists). For example, a telephone directory might map from last-name, first-name pairs to telephone numbers. Assuming that we have defined last, first and number, we could write:

> 把元组用作字典中的键是很常见的做法（主要也是因为这种情况不能用列表）。比如，一个电话字典可能就映射了姓氏、名字的数据对到不同的电话号码。假如我们定义了 last，first 和 number 这三个变量，可以用如下方法来实现：

```
directory[last, first] = number
```

The expression in brackets is a tuple. We could use tuple assignment to traverse this dictionary.

> 方括号内的表达式是一个元组。我们可以用元组赋值语句来遍历这个字典。

```
for last, first in directory:
    print(first, last, directory[last,first])
```

This loop traverses the keys in directory, which are tuples. It assigns the elements of each tuple to last and first, then prints the name and corresponding telephone number.

> 上面这个循环会遍历字典中的键，这些键都是元组。程序会把每个元组的元素分别赋值给 last 和 first，然后输出名字以及对应的电话号。

There are two ways to represent tuples in a state diagram. The more detailed version shows the indices and elements just as they appear in a list. For example, the tuple ('Cleese', 'John') would appear as in Figure 12.1.

> 在状态图中表示元组的方法有两种。更详尽的版本会展示索引和元素，就如同在列表中一样。例如图12.1中展示了元组('Cleese', 'John')。

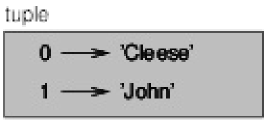


Figure 12.1: State diagram.

But in a larger diagram you might want to leave out the details. For example, a diagram of the telephone directory might appear as in Figure 12.2.

> 但随着图解规模变大，你也许需要省略掉一些细节。比如电话字典的图解可能会像图12.2所示。

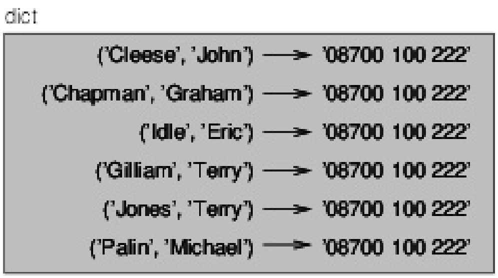


Figure 12.2: State diagram.

Here the tuples are shown using Python syntax as a graphical shorthand. The telephone number in the diagram is the complaints line for the BBC, so please don't call it.

> 图中的元组用 Python 的语法来简单表示。其中的电话号码是 BBC 的投诉热线，所以不要给人家打电话哈。

12.7 Sequences of sequences 序列的序列

I have focused on lists of tuples, but almost all of the examples in this chapter also work with lists of lists, tuples of tuples, and tuples of lists. To avoid enumerating the possible combinations, it is sometimes easier to talk about sequences of sequences.

> 之前我一直在讲由元组组成的列表，但本章几乎所有的例子也适用于由列表组成的列表、元组组成的元组以及列表组成的元组。为了避免枚举所有的组合，咱们直接讨论序列组成的序列就更方便一些。

In many contexts, the different kinds of sequences (strings, lists and tuples) can be used interchangeably. So how should you choose one over the others?

> 很多情况下，不同种类的序列（字符串、列表和元组）是可以交换使用的。那么该如何选择用哪种序列呢？

To start with the obvious, strings are more limited than other sequences because the elements have to be characters. They are also immutable. If you need the ability to change the characters in a string (as opposed to creating a new string), you might want to use a list of characters instead.

> 先从最简单的开始，字符串比起其他序列，功能更加有限，因为字符串中的元素必须是字符。而且还不能修改。如果你要修改字符串里面的字符（而不是要建立一个新字符串），你最好还是用字符列表吧。

Lists are more common than tuples, mostly because they are mutable. But there are a few cases where you might prefer tuples:

> 列表用的要比元组更广泛，主要因为列表可以修改。但以下这些情况下，你还是用元组更好：

1. In some contexts, like a return statement, it is syntactically simpler to create a tuple than a list.

> 在某些情况下，比如返回语句中，用元组来实现语法上要比列表简单很多。

1. If you want to use a sequence as a dictionary key, you have to use an immutable type like a tuple or string.

> 如果你要用一个序列作为字典的键，必须用元组或者字符串这样不可修改的类型才行。

1. If you are passing a sequence as an argument to a function, using tuples reduces the potential for unexpected behavior due to aliasing.

> 如果你要把一个序列作为参数传给一个函数，用元组能够降低由于别名使用导致未知情况而带来的风险。

Because tuples are immutable, they don’t provide methods like sort and reverse, which modify existing lists. But Python provides the built-in function sorted, which takes any sequence and returns a new list with the same elements in sorted order, and reversed, which takes a sequence and returns an iterator that traverses the list in reverse order.

> 由于元组是不可修改的，所以不提供 sort 和 reverse 这样的方法，这些方法都只能修改已经存在的列表。但 Python 提供了内置函数 sorted，该函数接收任意序列，然后返回一个把该序列中元素重新排序过的列表，另外还有个内置函数 reversed，接收一个序列然后返回一个以逆序迭代整个列表的迭代器。

12.8 Debugging 调试

Lists, dictionaries and tuples are examples of data structures; in this chapter we are starting to see compound data structures, like lists of tuples, or dictionaries that contain tuples as keys and lists as values. Compound data structures are useful, but they are prone to what I call shape errors; that is, errors caused when a data structure has the wrong type, size, or structure. For example, if you are expecting a list with one integer and I give you a plain old integer (not in a list), it won't work.

> 列表、字典以及元组，都是数据结构的一些样例；在本章我们开始见识这些复合的数据结构，比如由元组组成的列表，或者包含元组作为键而列表作为键值的字典等等。符合数据结构非常有用，但容易导致一些错误，我把这种错误叫做结构错误；这种错误往往是由于一个数据结构中出现了错误的类型、大小或者结构而引起的。比如，如果你想要一个由一个整形构成的列表，而我给你一个单纯的整形变量（不是放进列表的），就会出错了。

To help debug these kinds of errors, I have written a module called structshape that provides a function, also called structshape, that takes any kind of data structure as an argument and returns a string that summarizes its shape. You can download it from Here.

> 要想有助于解决这类错误，我写了一个叫做structshape 的模块，该模块提供了一个同名函数，接收任何一种数据结构作为参数，然后返回一个字符串来总结该数据结构的形态。可以从 这里下载。

Here's the result for a simple list:

> 下面是一个简单列表的示范：

```
>>> from structshape import structshape
>>> from structshape import structshape
>>> t = [1, 2, 3]
>>> t = [1, 2, 3]
>>> structshape(t)
>>> structshape(t)
'list of 3 int'
```

A fancier program might write "list of 3 ints", but it was easier not to deal with plurals. Here's a list of lists:

> 更带劲点的程序可能还应该写“list of 3 ints”，但不理会单复数变化有利于简化问题。下面是一个列表的列表：

```
>>> t2 = [[1,2], [3,4], [5,6]]
>>> t2 = [[1,2], [3,4], [5,6]]
>>> structshape(t2)
>>> structshape(t2)
'list of 3 list of 2 int'
```

If the elements of the list are not the same type, structshape groups them, in order, by type:

> 如果列表元素是不同类型，structshape 会按照顺序，把每种类型都列出：

```
>>> t3 = [1, 2, 3, 4.0, '5', '6', [7], [8], 9]
>>> t3 = [1, 2, 3, 4.0, '5', '6', [7], [8], 9]
>>> structshape(t3)
>>> structshape(t3)
'list of (3 int, float, 2 str, 2 list of int, int)'
```

Here’s a list of tuples:

> 下面是一个元组的列表：

```
>>> s = 'abc'
>>> s = 'abc'
>>> lt = list(zip(t, s))
>>> lt = list(zip(t, s))
>>> structshape(lt)
>>> structshape(lt)
'list of 3 tuple of (int, str)'
```

And here’s a dictionary with 3 items that map integers to strings.

> 下面是一个有三个项的字典，该字典映射了从整形数到字符串。

```
>>> d = dict(lt)
>>> d = dict(lt)
>>> structshape(d)
>>> structshape(d)
'dict of 3 int->str'
```

If you are having trouble keeping track of your data structures, structshape can help.

> 如果你追踪自己的数据结构有困难，structshape这个模块能有所帮助。

12.9 Glossary 术语列表

tuple: An immutable sequence of elements.

> 元组：一列元素组成的不可修改的序列。

tuple assignment: An assignment with a sequence on the right side and a tuple of variables on the left. The right side is evaluated and then its elements are assigned to the variables on the left.

> 元组赋值：一种赋值语句，等号右侧用一个序列，左侧为一个变量构成的元组。右侧的内容先进行运算，然后这些元素会赋值给左侧的变量。

gather: The operation of assembling a variable-length argument tuple.

> 收集：变量长度可变元组添加元素的运算。

scatter: The operation of treating a sequence as a list of arguments.

> 分散：将一个序列拆分成一系列参数组成的列表的运算。

zip object: The result of calling a built-in function zip; an object that iterates through a sequence of tuples.

> 拉链对象：调用内置函数 zip 得到的返回结果；一个遍历元组序列的对象。

iterator: An object that can iterate through a sequence, but which does not provide list operators and methods.

> 迭代器：迭代一个序列的对象，这种序列不能提供列表的运算和方法。

data structure: A collection of related values, often organized in lists, dictionaries, tuples, etc.

> 数据结构：一些有关系数据的集合体，通常是列表、字典或者元组等形式。

shape error: An error caused because a value has the wrong shape; that is, the wrong type or size.

> 结构错误：由于一个值有错误的结构而导致的错误；比如错误的类型或者大小。

12.10 Exercises 练习

Exercise 1 练习1

Write a function called most_frequent that takes a string and prints the letters in decreasing order of frequency. Find text samples from several different languages and see how letter frequency varies between languages. Compare your results with the tables at Here. Solution.

> 写一个名为most_frequent的函数，接收一个字符串，然后用出现频率降序来打印输出字母。找一些不同语言的文本素材，然后看看不同语言情况下字母的频率变化多大。然后用你的结果与这里的数据进行对比。样例代码。

Exercise 2 练习2

More anagrams!

> 更多变位词了！

1. Write a program that reads a word list from a file (see Section 9.1) and prints all the sets of words that are anagrams.

> 写一个函数，读取一个文件中的一个单词列表（参考9.1），然后输出所有的变位词。

Here is an example of what the output might look like:

> 下面是可能的输出样式的示范：

['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled'] ['retainers', 'ternaries'] ['generating', 'greatening'] ['resmelts', 'smelters', 'termless'] Hint: you might want to build a dictionary that maps from a collection of letters to a list of words that can be spelled with those letters. The question is, how can you represent the collection of letters in a way that can be used as a key?

> 提示：你也许可以建立一个字典，映射一个特定的字母组合到一个单词列表，单词列表中的单词可以用这些字母来拼写出来。那么问题来了，如何去表示这个字母的集合，才能让这个集合能用作字典的一个键？

1. Modify the previous program so that it prints the longest list of anagrams first, followed by the second longest, and so on.

> 修改一下之前的程序，让它先输出变位词列表中最长的，然后是其次长的，依此类推。

1. In Scrabble a “bingo” is when you play all seven tiles in your rack, along with a letter on the board, to form an eight-letter word. What collection of 8 letters forms the most possible bingos? Hint: there are seven. Solution.

> 在拼字游戏中，当你已经有七个字母的时候，再添加一个字母就能组成一个八个字母的单词，这就 TMD『bingo』 了（什么鬼东西？老外者拼字游戏就跟狗一样，翻着恶心死了）。然后哪八个字母组合起来最可能得到 bingo？提示：有七个。（简直就是狗一样的题目，麻烦死了，这里数据结构大家学会了就好了。）

Exercise 3 练习3

Two words form a “metathesis pair” if you can transform one into the other by swapping two letters; for example, “converse” and “conserve”. Write a program that finds all of the metathesis pairs in the dictionary. Hint: don’t test all pairs of words, and don’t test all possible swaps. Solution. Credit: This exercise is inspired by an example at Here.

> 两个单词，如果其中一个通过调换两个字母位置就能成为另外一个，就成了一个『交换对』。协议额函数来找到词典中所有的这样的交换对。提示：不用测试所有的词对，不用测试所有可能的替换方案。样例代码。 鸣谢：本练习受启发于这里的一个例子。

Exercise 4 练习4

Here's another Car Talk Puzzler:

> 接下来又是一个汽车广播字谜:

What is the longest English word, that remains a valid English word, as you remove its letters one at a time? Now, letters can be removed from either end, or the middle, but you can't rearrange any of the letters. Every time you drop a letter, you wind up with another English word. If you do that, you're eventually going to wind up with one letter and that too is going to be an English word—one that's found in the dictionary. I want to know what's the longest word and how many letters does it have?

> 一个英文单词，每次去掉一个字母，又还是一个正确的英文单词，这是什么词？
>
> 然后接下来字母可以从头去掉，也可以从末尾去掉，或者从中间，但不能重新排列其他字母。每次去掉一个字母，都会的到一个新的英文单词。然后最终会得到一个字母，也还是一个英文单词，这个单词也能在词典中找到。符合这样要求的单词有多少？最长的是哪个？

I'm going to give you a little modest example: Sprite. Ok? You start off with sprite, you take a letter off, one from the interior of the word, take the r away, and we're left with the word spite, then we take the e off the end, we're left with spit, we take the s off, we're left with pit, it, and I.

> 给你一个合适的小例子：Sprite。这个词就满足上面的条件。把 r 去掉了是 spite，去掉结尾的 e 是 spit，去掉 s 得到的是 pit，it，然后是 I。

Write a program to find all words that can be reduced in this way, and then find the longest one. This exercise is a little more challenging than most, so here are some suggestions:

> 写一个函数找到所有的这样的词，然后找到其中最长的一个。
>
> 这个练习比一般的练习难以些，所以下面是一些提示：

1. You might want to write a function that takes a word and computes a list of all the words that can be formed by removing one letter. These are the "children" of the word.

> 你也许需要写一个函数，接收一个单词然后计算一下这个单词去掉一个字母能得到单词组成的列表。列表中这些单词就如同原始单词的孩子一样。

1. Recursively, a word is reducible if any of its children are reducible. As a base case, you

can consider the empty string reducible.

> 只要一个 单词的孩子还可以缩减，那这个单词本身就亏缩减。你可以认为空字符串是可以缩减的，这样来作为一个基准条件。

1. The wordlist I provided, words.txt, doesn’t contain single letter words. So you might want to add “I”, “a”, and the empty string.

> 我上面提供的 words.txt 这个词表，不包含单个字母的单词。所以你需要自行添加 I、a 以及空字符串上去。

1. To improve the performance of your program, you might want to memoize the words that are known to be reducible.

> 要提高程序性能的话，你最好存储住已经算出来能被继续缩减的单词。

Solution.

> 样例代码。

Chapter 13 Case study: data structure selection 案例学习：数据结构的选择

At this point you have learned about Python’s core data structures, and you have seen some of the algorithms that use them. If you would like to know more about algorithms, this might be a good time to read Chapter 13. But you don’t have to read it before you go on; you can read it whenever you are interested. This chapter presents a case study with exercises that let you think about choosing data structures and practice using them.

> 到现在位置，你已经学过 Python 中最核心的数据结构了，也学过了一些与之对应的各种算法了。如果你想要对算法进行深入的了解，就可以来读一下第十三章。但不读这一章也可以继续；无论什么时候读都可以，感兴趣了就来看即可。
>
> 本章通过一个案例和一些练习，来讲解一下如何选择和使用数据结构。

13.1 Word frequency analysis 词频统计

As usual, you should at least attempt the exercises before you read my solutions.

> 跟往常一样，你最起码也得先自己尝试做一下这些练习，然后再看参考答案。

Exercise 1 练习1

Write a program that reads a file, breaks each line into words, strips whitespace and punctuation from the words, and converts them to lowercase.

> 写一个读取文件的程序，把每一行拆分成一个个词，去掉空白和标点符号，然后把所有单词都转换成小写字母的。

Hint: The string module provides a string named whitespace, which contains space, tab, newline, etc., and punctuation which contains the punctuation characters. Let’s see if we can make Python swear:

> 提示：字符串模块 string 提供了一个名为 whitespace 的字符串，包含了空格、跳表符、另起一行等等，然后还有个 punctuation 模块，包含了各种标点符号的字符。咱们可以试试让 Python 把标点符号都给显示一下：

```
>>> import string
>>> import string
>>>string.punctuation
>>>string.punctuation
'!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~'
```

Also, you might consider using the string methods strip, replace and translate.

> 另外你也可以试试字符串模块的其他方法，比如 strip、replace 以及 translate。

Exercise 2 练习2

Go to [Project Gutenberg] (http://gutenberg.org) and download your favorite out-of-copyright book in plain text format.

> 访问[古登堡计划网站] (http://gutenberg.org)，然后下载一个你最喜欢的公有领域的书，要下载纯文本格式的哈。

Modify your program from the previous exercise to read the book you downloaded, skip over the header information at the beginning of the file, and process the rest of the words as before.

> 修改一下刚才上一个练习你写的程序，让这个程序能读取你下载的这本书，跳过文件开头部分的信息，继续以上个练习中的方式来处理一下整本书的正文。

Then modify the program to count the total number of words in the book, and the number of times each word is used.

> 然后再修改一下程序，让程序能统计一下整本书的总单词数目，以及每个单词出现的次数。

Print the number of different words used in the book. Compare different books by different authors, written in different eras. Which author uses the most extensive vocabulary?

> 输出一下这本书中不重复的单词的个数。对比一下不同作者、不同地域的书籍。哪个作者的词汇量最丰富？

Exercise 3 练习3

Modify the program from the previous exercise to print the 20 most frequently-used words in the book.

> 再接着修改程序，输出一下每本书中最频繁出现的20个词。

Exercise 4 练习4

Modify the previous program to read a word list (see Section 9.1) and then print all the words in the book that are not in the word list. How many of them are typos? How many of them are common words that should be in the word list, and how many of them are really obscure?

> 接着修改，让程序能读取一个单词列表（参考9.1），然后输出一下所有包含在书中，但不包含于单词列表中的单词。看看这些单词中有多少是排版错误的？有多少是本应被单词列表包含的常用单词？有多少是很晦涩艰深的罕见词汇？

13.2 Random numbers 随机数

Given the same inputs, most computer programs generate the same outputs every time, so they are said to be deterministic. Determinism is usually a good thing, since we expect the same calculation to yield the same result. For some applications, though, we want the computer to be unpredictable. Games are an obvious example, but there are more.

> 输入相同的情况下，大多数计算机程序每次都会给出相同的输出，这也叫做确定性。确定性通常是一件好事，因为我们都希望同样的运算产生同样的结构。但有时候为了一些特定用途，咱们就需要让计算机能有不可预测性。比如游戏等等，有很多很多这样的情景。

Making a program truly nondeterministic turns out to be difficult, but there are ways to make it at least seem nondeterministic. One of them is to use algorithms that generate pseudorandom numbers. Pseudorandom numbers are not truly random because they are generated by a deterministic computation, but just by looking at the numbers it is all but impossible to distinguish them from random.

> 然而，想让一个程序真正变成不可预测的，也是很难的，但好在有办法能让程序看上去不太确定。其中一种方法就是通过算法来产生假随机数。假随机数，顾名思义就知道不是真正随机的了，因为它们是通过一种确定性的运算来得到的，但这些数字看上去是随机的，很难与真正的随机数区分。
>
> （译者注：这里大家很容易一带而过，而不去探究到底怎样能确定是真随机数。实际上随机数是否能得到以及是否存在会影响哲学判断，可知论和不可知论等等。那么就建议大家思考和搜索一下，随机数算法产生的随机数和真正随机数有什么本质的区别，以及是否有办法得到真正的随机数。如果有，如何得到呢？）

The random module provides functions that generate pseudorandom numbers (which I will simply call “random” from here on).

> random 模块提供了生成假随机数的函数（从这里开始，咱们就用随机数来简称假随机数了哈）。

The function random returns a random float between 0.0 and 1.0 (including 0.0 but not 1.0). Each time you call random, you get the next number in a long series. To see a sample, run this loop:

> 函数 random 返回一个在0.0到1.0的前闭后开区间（就是包括0.0但不包括1.0，这个特性在 Python 随处都是，比如序列的索引等等）的随机数。每次调用 random，就会得到一个很长的数列中的下一个数。如下这个循环就是一个例子了：

```
import random  for i in range(10):
x = random.random()
print(x)
```

The function randint takes parameters low and high and returns an integer between low and high (including both).

> randint函数接收两个参数作为下界和上界，然后返回一个二者之间的整数，这个整数可以是下界或者上界。

```
>>> random.randint(5, 10)
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
>>> random.randint(5, 10)
9
```

To choose an element from a sequence at random, you can use choice:

> choice 函数可以用来从一个序列中随机选出一个元素：

```
>>> t = [1, 2, 3]
>>> t = [1, 2, 3]
>>> random.choice(t)
>>> random.choice(t)
2
>>> random.choice(t)
>>> random.choice(t)
3
```

The random module also provides functions to generate random values from continuous distributions including Gaussian, exponential, gamma, and a few more.

> random 模块还提供了其他一些函数，可以计算某些连续分布的随机值，比如Gaussian高斯分布, exponential指数分布, gamma γ分布等等。

Exercise 5 练习5

Write a function named choose_from_hist that takes a histogram as defined in Section 11.2 and returns a random value from the histogram, chosen with probability in proportion to frequency. For example, for this histogram:

> 写一个名为 choose_from_hist 的函数，用这个函数来处理一下11.2中定义的那个histogram函数，从histogram 的值当中随机选择一个，这个选择的概率按照比例来定。比如下面这个histogram：

```
>>> t = ['a', 'a', 'b']
>>> t = ['a', 'a', 'b']
>>> hist = histogram(t)
>>> hist = histogram(t)
>>> hist
>>> hist
{'a': 2, 'b': 1}
```

your function should return 'a' with probability 2/3 and 'b' with probability 1/3.

> 你的函数就应该返回a 的概率为2/3，返回b 的概率为1/3

13.3 Word histogram 词频

You should attempt the previous exercises before you go on. You can download my solution from Here.

> 你得先把前面的练习作一下，然后再继续哈。可以从这里下载我的样例代码。

You will also need This.

> 此外还要下载这个。

Here is a program that reads a file and builds a histogram of the words in the file:

> 下面这个程序先读取一个文件，然后对该文件中的词频进行了统计：

```
import string
def process_file(filename):
    hist = dict()
    fp = open(filename)
    for line in fp:
        process_line(line, hist)
    return hist
def process_line(line, hist):
    line = line.replace('-', ' ')
    for word in line.split():
        word = word.strip(string.punctuation + string.whitespace)
        word = word.lower()
        hist[word] = hist.get(word, 0) + 1
hist = process_file('emma.txt')
```

This program reads emma.txt, which contains the text of Emma by Jane Austen.

> 上面这个程序读取的是 emma.txt 这个文件，该文件是简奥斯汀的小说《艾玛》。

process_file loops through the lines of the file, passing them one at a time to process_line. The histogram hist is being used as an accumulator.

> process_file这个函数遍历整个文件，逐行读取，然后把每行的内容发给process_line函数。词频统计函数 hist 在该程序中是一个累加器。

process_line uses the string method replace to replace hyphens with spaces before using split to break the line into a list of strings. It traverses the list of words and uses strip and lower to remove punctuation and convert to lower case. (It is a shorthand to say that strings are “converted”; remember that string are immutable, so methods like strip and lower return new strings.)

> process_line使用字符串的方法 replace把各种连字符都用空格替换，然后用 split 方法把整行打散成一个字符串列表。程序遍历整个单词列表，然后用 strip 和 lower 这两个方法移除了标点符号，并且把所有字母都转换成小写的。（一定要记住，这里说的『转换』是图方便而已，实际上并不能转换，要记住字符串是不可以修改的，strip 和 lower 这些方法都是返回了新的字符串，一定要记得！）

Finally, process_line updates the histogram by creating a new item or incrementing an existing one. To count the total number of words in the file, we can add up the frequencies in the histogram:

> 最终，process_line 函数通过建立新项或者累加已有项，对字频统计 histogram 进行了更新。
>
> 要计算整个文件中的单词总数，就可以把 histogram 中的所有频数加到一起就可以了：

```
def total_words(hist):
    return sum(hist.values())
```

The number of different words is just the number of items in the dictionary:

> 不重复的单词的数目也就是字典中项的个数了：

```
def different_words(hist):
    return len(hist)
```

Here is some code to print the results:

> 输出结果的代码如下：

```
print('Total number of words:', total_words(hist))
print('Number of different words:', different_words(hist))
```

And the results:

> 结果如下所示：

```
Total number of words: 161080
Number of different words: 7214
```

13.4 Most common words 最常用的单词

To find the most common words, we can make a list of tuples, where each tuple contains a word and its frequency, and sort it. The following function takes a histogram and returns a list of word-frequency tuples:

> 要找到最常用的词，可以做一个元组列表，每一个元组包含一个单词和该单词出现的次数，然后整理一下这个列表，就可以了。
>
> 下面的函数就接收了词频统计结果，然后返回一个『单词-次数』元组组成的列表：

```
def most_common(hist):
    t = []
    for key, value in hist.items():
        t.append((value, key))
        t.sort(reverse=True)
    return t
```

In each tuple, the frequency appears first, so the resulting list is sorted by frequency. Here is a loop that prints the ten most common words:

> 这些元组中，要先考虑词频，返回的列表因此根据词频来排序。下面是一个输出最常用单词的循环体：

```
t = most_common(hist)
print('The most common words are:')
for freq, word in t[:10]:
    print(word, freq, sep='\t')
```

I use the keyword argument sep to tell print to use a tab character as a “separator”, rather than a space, so the second column is lined up. Here are the results from Emma:

> 此处用了关键词 sep 来让 print 输出的时候以一个tab跳表符来作为分隔，而不是一个空格，这样第二列就会对齐。下面就是对《艾玛》这本小说的统计结果：
>
> (译者注：这个效果在 Python 下很明显，此处 markdown 我刚开始熟悉，不清楚咋实现。）

```
The most common words are:
to      5242
the     5205
and     4897
of      4295
i       3191
a       3130
it      2529
her     2483
was     2400
she     2364
```

This code can be simplified using the key parameter of the sort function. If you are curious, you can read about it at Here.

> 如果使用 sort 函数的 key 参数，上面的代码还可以进一步简化。如果你好奇的话，可以进一步阅读一下说明

13.5 Optional parameters 可选的参数

We have seen built-in functions and methods that take optional arguments. It is possible to write programmer-defined functions with optional arguments, too. For example, here is a function that prints the most common words in a histogram:

> 咱们已经看过好多有可选参数的内置函数和方法了。实际上咱们自己也可以写，写这种有可选参数的自定义函数。比如下面就是一个根据词频数据来统计最常用单词的函数：

```
def print_most_common(hist, num=10):
    t = most_common(hist)
    print('The most common words are:')
    for freq, word in t[:num]:
        print(word, freq, sep='\t')
```

The first parameter is required; the second is optional. The default value of num is 10. If you only provide one argument:

> 上面这个函数中，第一个参数是必须输入的；而第二个参数就是可选的了。第二个参数 num 的默认值是10.
>
> 如果只提供第一个参数：

```
print_most_common(hist)
```

num gets the default value. If you provide two arguments:

> 这样 num 就用默认值了。如果提供两个参数：

```
print_most_common(hist, 20)
```

num gets the value of the argument instead. In other words, the optional argument overrides the default value. If a function has both required and optional parameters, all the required parameters have to come first, followed by the optional ones.

> 这样 num 就用参数值来赋值了。换句话说，可选参数可以覆盖默认值。
>
> 如果一个函数同时含有必需参数和可选参数，就必须在定义函数的时候，把必需参数全都放到前面，而可选的参数要放到后面。

13.6 Dictionary subtraction 字典减法

Finding the words from the book that are not in the word list from words.txt is a problem you might recognize as set subtraction; that is, we want to find all the words from one set (the words in the book) that are not in the other (the words in the list).

> 有的单词存在于书当中，但没有包含在文件 words.txt 的单词列表中，找这些单词就有点难了，你估计已经意识到了，这是一种集合的减法；也就是要从一个集合（也就是书）中所有不被另一个集合（也就是单词列表）包含的单词。

subtract takes dictionaries d1 and d2 and returns a new dictionary that contains all the keys from d1 that are not in d2. Since we don't really care about the values, we set them all to None.

> 下面的代码中定义的 subtrac t这个函数，接收两个字典 d1和 d2，然后返回一个新字典，这个新字典包含所有 d1中包含而 d2中不包含的键。键值就无所谓了，就都设置为空即可。

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

To find the words in the book that are not in words.txt, we can use process_file to build a histogram for words.txt, and then subtract:

> 要找到书中含有而words.txt 中不含有的单词，就可以用 process_file 函数来建立一个 words.txt 的词频统计，然后用 subtract 函数来相减：

```
words = process_file('words.txt')
diff = subtract(hist, words)
print("Words in the book that aren't in the word list:")
for word in diff.keys():
    print(word, end=' ')
```

Here are some of the results from Emma:

> 下面依然还是对《艾玛》得到的结果：

```
Words in the book that aren't in the word list:
rencontre
jane's
blanche
woodhouses
disingenuousness
friend's
venice
apartment
...
```

Some of these words are names and possessives. Others, like “rencontre”, are no longer in common use. But a few are common words that should really be in the list!

> 这些单词有的是名字或者所有格之类的。另外的一些，比如『rencontre』，都是现在不怎么常用的了。不过也确实有一些单词是挺常用的，挺应该被列表所包含的！

Exercise 6 练习6

Python provides a data structure called set that provides many common set operations. You can read about them in Section 19.5, or read the documentation at Here.

> Python 提供了一个数据结构叫 set（集合），该类型提供了很多常见的集合运算。可以在19.5阅读一下，或者阅读一下这里的官方文档。

Write a program that uses set subtraction to find words in the book that are not in the word list. Solution.

> 写一个程序吧，用集合的减法，来找一下书中包含而列表中不包含的单词吧。 样例代码。

13.7 Random words 随机单词

To choose a random word from the histogram, the simplest algorithm is to build a list with multiple copies of each word, according to the observed frequency, and then choose from the list:

> 要从词频数据中选一个随机的单词，最简单的算法就是根据已知的单词频率来将每个单词复制相对应的个数的副本，然后组建成一个列表，从列表中选择单词：

```
def random_word(h):
    t = []
    for word, freq in h.items():
        t.extend([word] * freq)
    return random.choice(t)
```

The expression [word] * freq creates a list with freq copies of the string word. The extend method is similar to append except that the argument is a sequence.

> 上面代码中的[word] * freq表达式建立了一个列表，列表中字符串单词的出现次数即其原来的词频数。extend 方法和 append 方法相似，区别是前者的参数是一个序列，而后者是单独的元素。

This algorithm works, but it is not very efficient; each time you choose a random word, it rebuilds the list, which is as big as the original book. An obvious improvement is to build the list once and then make multiple selections, but the list is still big.

> 上面这个算法确实能用，但效率实在不怎么好；每次选择随机单词的时候，程序都要重建列表，这个列表就和源书一样大了。很显然，一次性建立列表，而多次使用该列表来进行选择，这样能有明显改善，但列表依然还是很大。

An alternative is:

> 备选的思路如下：

1. Use keys to get a list of the words in the book.

> 用键来存储书中单词的列表。

1. Build a list that contains the cumulative sum of the word frequencies (see Exercise 2). The last item in this list is the total number of words in the book, n.

> 再建立一个列表，该列表包含所有词频的累加总和（参考练习2）。该列表的最后一个元素是书中所有单词的总数 n。

1. Choose a random number from 1 to n. Use a bisection search (See Exercise 10) to find the index where the random number would be inserted in the cumulative sum.

> 选择一个1到 n 之间的随机数。使用折半法搜索（参考练习10），找到随机数在累计总和中所在位置的索引值。

1. Use the index to find the corresponding word in the word list.

> 用该索引值来找到单词列表中对应的单词。

Exercise 7 练习7

Write a program that uses this algorithm to choose a random word from the book. Solution.

> 写一个程序，用上面说的算法来从一本书中随机挑选单词。样例代码。

13.8 Markov analysis 马科夫分析法

If you choose words from the book at random, you can get a sense of the vocabulary, but you probably won't get a sentence:

> 如果让你从一本书中随机挑选一些单词，这些单词都能理解，但估计难以成为一句话：

```
this the small regard harriet which knightley's it most things
```

A series of random words seldom makes sense because there is no relationship between successive words. For example, in a real sentence you would expect an article like “the” to be followed by an adjective or a noun, and probably not a verb or adverb.

> 一系列随机次很少能组成整句的话，因为这些单词连接起来并没有什么关系。例如，成句的话中，冠词 the 后面应该是跟着形容词或者名词，而不应该是动词或者副词。

One way to measure these kinds of relationships is Markov analysis, which characterizes, for a given sequence of words, the probability of the words that might come next. For example, the song Eric, the Half a Bee begins:

> 衡量单词之间关系的一种方法就是马科夫分析法，这一方法就是：对给定的单词序列，分析一个词跟着另一个词后面出现的概率。比如，Eric, the Half a Bee这首歌的开头：

```
Half a bee, philosophically,
Must, ipso facto, half not be.
But half the bee has got to be
Vis a vis, its entity. D’you see?
But can a bee be said to be
Or not to be an entire bee
When half the bee is not a bee
Due to some ancient injury?
```

In this text, the phrase “half the” is always followed by the word “bee”, but the phrase “the bee” might be followed by either “has” or “is”.

> 在上面的文本中，『half the』这个词组后面总是跟着『bee』，但词组『the bee』后面可以是『has』，也可以是『is』。

The result of Markov analysis is a mapping from each prefix (like “half the” and “the bee”) to all possible suffixes (like “has” and “is”).

> 马科夫分析的结果是从每个前缀（比如『half the』和『the bee』）到所有可能的后缀（比如『has』和『is』）的映射。

Given this mapping, you can generate a random text by starting with any prefix and choosing at random from the possible suffixes. Next, you can combine the end of the prefix and the new suffix to form the next prefix, and repeat.

> 有了这一映射，你就可以制造随机文本了，用任意的前缀开头，然后从可能的后缀中随机选一个。下一次就把前缀的末尾和新的后缀结合起来，作为新的前缀，然后重复上面的步骤。

For example, if you start with the prefix “Half a”, then the next word has to be “bee”, because the prefix only appears once in the text. The next prefix is “a bee”, so the next suffix might be “philosophically”, “be” or “due”. In this example the length of the prefix is always two, but you can do Markov analysis with any prefix length.

> 例如，你用前缀『Half a』来开始，那接下来的就必须是『bee』了，因为这个前缀只在文本中出现了一次。接下来，第二次了，前缀就变成了『a bee』了，所以接下来的后缀可以是『philosophically』，『be』或者『due』。
>
> 在这个例子中，前缀的长度总是两个单词，但你可以以任意长度的前缀来进行马科夫分析。

Exercise 8 练习8

Markov analysis:

> 马科夫分析：

1. Write a program to read a text from a file and perform Markov analysis. The result should be a dictionary that maps from prefixes to a collection of possible suffixes. The collection might be a list, tuple, or dictionary; it is up to you to make an appropriate choice. You can test your program with prefix length two, but you should write the program in a way that makes it easy to try other lengths.

> 写一个程序，读取文件中的文本，然后进行马科夫分析。结果应该是一个字典，从前缀到一个可能的后缀组成的序列的映射。这个序列可以是列表，元组，也可以是字典；你自己来选择合适的类型来写就好。你可以用两个单词长度的前缀来测试你的程序，但应该让程序能够兼容其他长度的前缀。

1. Add a function to the previous program to generate random text based on the Markov analysis. Here is an example from Emma with prefix length 2:

> 在上面的程序中添加一个函数，基于马科夫分析来生成随机文本。下面是用《艾玛》使用两个单词长度的前缀来生成的一个随机文本样例：

```
He was very clever, be it sweetness or be angry, ashamed or only amused, at such a stroke
```

For this example, I left the punctuation attached to the words. The result is almost syntactically correct, but not quite. Semantically, it almost makes sense, but not quite.

> 这个例子中，我保留了单词中连接的标点符号。得到的结果在语法上基本是正确的，但也还差点。语义上，这些单词连起来也还能有一些意义，但也不咋对劲。

What happens if you increase the prefix length? Does the random text make more sense?

> 如果增加前缀的单词长度会怎么样？随机文本是不是读起来更通顺呢？

1. Once your program is working, you might want to try a mash-up: if you combine text from two or more books, the random text you generate will blend the vocabulary and phrases from the sources in interesting ways.

> 一旦你的程序能用了，你可以试试混搭一下：如果你把两本以上的书合并起来，生成的随机文本就会以很有趣的方式从多种来源混合单词和短语来生成随机文本。

Credit: This case study is based on an example from Kernighan and Pike, The Practice of Programming, Addison-Wesley, 1999.

> 引用：这个案例研究是基于Kernighan 和 Pike 在1999年由Addison-Wesley出版社出版的《The Practice of Programming》一书中的一个例子。

You should attempt this exercise before you go on; then you can can download my solution from Here. You will also need This, the txt file of Emma.

> 你应该自己独立尝试一下这些练习，然后再继续；然后你可以下载我的样例代码。另外你可能需要下载 《艾玛》这本书的文本文件。

13.9 Data structures 数据结构

Using Markov analysis to generate random text is fun, but there is also a point to this exercise: data structure selection. In your solution to the previous exercises, you had to choose:

> 使用马科夫分析来生成随机文本挺有意思的，但这个练习还有另外一个要点：数据结构的选择。在前面这些练习中，你必须要选择以下内容：

- How to represent the prefixes.

> 如何表示前缀。

- How to represent the collection of possible suffixes.

> 如何表示可能后缀的集合。

- How to represent the mapping from each prefix to the collection of possible suffixes.

> 如何表示每个前缀与对应的后缀集合之间的映射。

The last one is easy: a dictionary is the obvious choice for a mapping from keys to corresponding values.

> 最后一个最简单了：明显就应该用字典了，这样来把每个键映射到对应的多个值。

For the prefixes, the most obvious options are string, list of strings, or tuple of strings.

> 前缀的选择，明显可以使用字符串、字符串列表，或者字符串元组。

For the suffixes, one option is a list; another is a histogram (dictionary).

> 后缀的先泽，要么是用列表，要么就用咱们之前写过的词频函数 histogram（这个也是个字典）。

How should you choose? The first step is to think about the operations you will need to implement for each data structure. For the prefixes, we need to be able to remove words from the beginning and add to the end. For example, if the current prefix is “Half a”, and the next word is “bee”, you need to be able to form the next prefix, “a bee”.

> 该咋选呢？第一步就是想一下，每种数据结构都需要用到哪些运算。比如对前缀来说，咱们就得能删掉头部，然后在尾部添加新词。例如，加入现在的前缀是『Half a』，接下来的单词是『bee』，就得能够组成下一个前缀，也就是『a bee』。

Your first choice might be a list, since it is easy to add and remove elements, but we also need to be able to use the prefixes as keys in a dictionary, so that rules out lists. With tuples, you can’t append or remove, but you can use the addition operator to form a new tuple:

> 你的首选估计就是列表了，因为列表很容易增加和剔除元素，但我们还需要能用前缀做字典中的键，所以列表就不合格了。那就剩元组了，元组没法添加和删除元素，但可以用加法运算符来建立新的元组。

```
def shift(prefix, word):
    return prefix[1:] + (word,)
```

shift takes a tuple of words, prefix, and a string, word, and forms a new tuple that has all the words in prefix except the first, and word added to the end.

> 上面这个 shift 函数，接收一个单词的元组，也就是前缀，然后还接收一个字符串，也就是单词了，然后形成一个新的元组，就是把原有的前缀去掉头部，用新单词拼接到尾部。

For the collection of suffixes, the operations we need to perform include adding a new suffix (or increasing the frequency of an existing one), and choosing a random suffix.

> 对后缀的集合来说，我们需要进行的运算包括添加新的后缀（或者增加一个已有后缀的频次），然后选择一个随机的后缀。

Adding a new suffix is equally easy for the list implementation or the histogram. Choosing a random element from a list is easy; choosing from a histogram is harder to do efficiently (see Exercise 7).

> 添加新后缀，无论是用列表还是用词频字典，实现起来都一样容易。不过从列表中选择一个随机元素很容易；但从词频字典中选择随机元素实现起来就不太有效率了（参考练习7）。

So far we have been talking mostly about ease of implementation, but there are other factors to consider in choosing data structures. One is run time. Sometimes there is a theoretical reason to expect one data structure to be faster than other; for example, I mentioned that the in operator is faster for dictionaries than for lists, at least when the number of elements is large.

> 目前为止，我们说完了实现难度，但对数据结构的选择还要考虑一些其他的因素。比如运行时间。有时候要考虑理论上的原因来考虑，最好的数据结构要比其他的快；例如我之前提到了 in 运算符在字典中要比列表中速度快，最起码当元素数量增多的时候会体现出来。

But often you don't know ahead of time which implementation will be faster. One option is to implement both of them and see which is better. This approach is called benchmarking. A practical alternative is to choose the data structure that is easiest to implement, and then see if it is fast enough for the intended application. If so, there is no need to go on. If not, there are tools, like the profile module, that can identify the places in a program that take the most time.

> 但一般情况下，咱们不能提前知道哪一种实现方法的速度更快。所以就可以两种都写出来，然后运行对比一下，看看到底哪个快。这种方法就叫对比测试。另外一种方法是选一个实现起来最简单的数据结构，然后看看运行速度是不是符合问题的要求。如果可以，就不用再改进了。如果速度不够快，就亏用到一些工具，比如 profile 模块，来判断程序的哪些部分消耗了最多的运行时间。

The other factor to consider is storage space. For example, using a histogram for the collection of suffixes might take less space because you only have to store each word once, no matter how many times it appears in the text. In some cases, saving space can also make your program run faster, and in the extreme, your program might not run at all if you run out of memory. But for many applications, space is a secondary consideration after run time.

> 此外还要考虑的一个因素就是存储空间了。比如用一个词频字典作为后缀集合就可能省一些存储空间，因为无论这些单词在稳重出现了多少次，在该字典中每个单词只存储一次。有的情况下，节省空间也能让你的程序运行更快，此外在一些极端情况下，比如内存耗尽了，你的程序就根本无法运行了。不过对大多数应用来说，都是优先考虑运行时间，存储空间只是次要因素了。

One final thought: in this discussion, I have implied that we should use one data structure for both analysis and generation. But since these are separate phases, it would also be possible to use one structure for analysis and then convert to another structure for generation. This

would be a net win if the time saved during generation exceeded the time spent in conversion.

> 最后再考虑一下：上文中，我已经暗示了，咱们选择某种数据结构，要兼顾分析和生成。但这二者是分开的步骤，所以也可以分析的时候用一种数据结构，而生成的时候再转换成另外一种结构。只要生成时候节省的时间胜过转换所花费的时间，相权衡之下依然是划算的。

13.10 Debugging 调试

When you are debugging a program, and especially if you are working on a hard bug, there are five things to try:

> 调试一个程序的时候，尤其是遇到特别严峻的问题的时候，有以下五个步骤一定要做好：

* Reading:

> 阅读代码：

Examine your code, read it back to yourself, and check that it says what you meant to say.

> 好好检查代码，多读几次，好好看看代码所表述的内容是不是跟你的设想相一致。

* Running:

> 运行程序：

Experiment by making changes and running different versions. Often if you display the right thing at the right place in the program, the problem becomes obvious, but sometimes you have to build scaffolding.

> 做一些修改，然后运行各个版本来对比实验一下。通常来说，只要你在程序对应的位置加上输出，问题就能比较明确了，不过有时候你还是得搭建一些脚手架代码来帮忙找错误。

* Ruminating:

> 反复思考：

Take some time to think! What kind of error is it: syntax, runtime, or semantic? What information can you get from the error messages, or from the output of the program? What kind of error could cause the problem you're seeing? What did you change last, before the problem appeared?

> 多花点时间去思考！想下到底是什么类型的错误：语法，运行，还是语义错误？从错误信息以及程序的输出能得到什么信息？想想哪种错误能引起你所看到的问题？问题出现之前的那一次你做了什么修改？

- Rubberducking:

> 小黄鸭调试法：

If you explain the problem to someone else, you sometimes find the answer before you finish asking the question. Often you don't need the other person; you could just talk to a rubber duck. And that's the origin of the well-known strategy called rubber duck debugging. I am not making this up; see Here.

> 如果你对另外一个人解释问题，你有时候就能在问完问题之前就找到答案。通常你根本不用找另外一个人；就根一个橡胶鸭子说就可以了。这就是很著名的所谓小黄鸭调试法的起源了。我可不是瞎编的哈；看这里的解释。

- Retreating:

> 以退为进：

At some point, the best thing to do is back off, undoing recent changes, until you get back to a program that works and that you understand. Then you can start rebuilding.

> 有时候，最佳的策略反而就是后撤，取消最近的修改，一直到程序恢复工作，并且你能清楚理解。然后再重头来改进。

Beginning programmers sometimes get stuck on one of these activities and forget the others. Each activity comes with its own failure mode.

> 新手程序员经常会在上面这些步骤中的某一项上卡壳，然后忘了其他的步骤。上面的每一步都有各自的失灵情况。

For example, reading your code might help if the problem is a typographical error, but not if the problem is a conceptual misunderstanding. If you don't understand what your program does, you can read it 100 times and never see the error, because the error is in your head.

> 比如，错误很典型的情况下，阅读代码也许有效，但如果错误是概念上误解导致的，这就没啥用了。如果你不理解你程序的功能，你就算读上一百测也找不到错误，因为是你脑中的理解有错误。

Running experiments can help, especially if you run small, simple tests. But if you run experiments without thinking or reading your code, you might fall into a pattern I call "random walk programming", which is the process of making random changes until the program does the right thing. Needless to say, random walk programming can take a long time.

> 在你进行小规模的简单测试的时候，进行试验会有用。但如果不思考和阅读代码，你就可以陷入到我称之为『随机走路编程』的陷阱中，这种过程就是随机做一些修改，一直到程序工作位置。毋庸置疑，这种随机修改肯定得浪费好多时间的。

You have to take time to think. Debugging is like an experimental science. You should have at least one hypothesis about what the problem is. If there are two or more possibilities, try to think of a test that would eliminate one of them.

> 最重要的就是思考，一定要花时间去思考。调试就像是一种实验科学。你至少应该对问题的本质有一种假设。如果有两种或者两种以上的可能性，就要设计个测试，来逐个排除可能性。

But even the best debugging techniques will fail if there are too many errors, or if the code you are trying to fix is too big and complicated. Sometimes the best option is to retreat, simplifying the program until you get to something that works and that you understand.

> 然而一旦错误特别多了，再好的调试技术也不管用的，程序太大太复杂也会容易有类似情况。所以有时候最好的方法就是以退为进，简化一下程序，直到能工作了，并且你能理解整个程序了为止。

Beginning programmers are often reluctant to retreat because they can't stand to delete a line of code (even if it's wrong). If it makes you feel better, copy your program into another file before you start stripping it down. Then you can copy the pieces back one at a time.

> 新手程序员经常不愿意后撤，因为他们不情愿删掉一行代码（哪怕是错误的代码）。可以这样，复制一下整个代码到另外一个文件中做个备份，然后再删减，这样是不是感觉好些。然后你可以再复制回来的。

Finding a hard bug requires reading, running, ruminating, and sometimes retreating. If you get stuck on one of these activities, try the others.

> 找到一个困难问题的解决方法，需要阅读、测试、分析，有时候还要后撤。如果你在某一步骤中卡住了，试试其他方法。

13.11 Glossary 术语列表

deterministic: Pertaining to a program that does the same thing each time it runs, given the same inputs.

> 确定性：给定同样的输出，程序每次运行结果都相同。

pseudorandom: Pertaining to a sequence of numbers that appears to be random, but is generated by a deterministic program.

> 假随机数：一段数字序列中的数，看上去似乎是随机的，但实际上也是由确定的算法来生成的。

default value: The value given to an optional parameter if no argument is provided.

> 默认值：如果不对可选参数进行赋值的话，该参数会用默认设置的值。

override: To replace a default value with an argument.

> 覆盖：用户在调用函数的时候给可选参数提供了参数，这个参数就覆盖掉默认值。

benchmarking: The process of choosing between data structures by implementing alternatives and testing them on a sample of the possible inputs.

> 对比测试：

rubber duck debugging: Debugging by explaining your problem to an inanimate object such as a rubber duck. Articulating the problem can help you solve it, even if the rubber duck doesn't know Python.

> 小黄鸭调试法：对一个无生命的对象来解释你的问题，比如小黄鸭之类的，这样来调试。描述清楚问题很有助于解决问题，所以虽然小黄鸭并不会理解 Python 也不要紧。

13.12 Exercises 练习

Exercise 9 练习9

The "rank" of a word is its position in a list of words sorted by frequency: the most common word has rank 1, the second most common has rank 2, etc.

> 单词的『排名』就是在一个单词列表中，按照出现频率而排的位置：最常见的单词就排名第一了，第二常见的就排第二，依此类推。

Zipf's law describes a relationship between the ranks and frequencies of words in natural languages . Specifically, it predicts that the frequency, f, of the word with rank r is:

> Zipf定律 描述了自然语言中排名和频率的关系。该定律预言了排名 r 与词频 f 之间的关系如下：

```
f = cr^{-s}
```

where s and c are parameters that depend on the language and the text. If you take the logarithm of both sides of this equation, you get:

> 这里的 s 和 c 都是参数，依据语言和文本而定。如果对等式两边同时取对数，得到如下公式：

```
\log f = \log c − s*\log r
```

> （译者注：Zipf定律是美国学者G.K.齐普夫提出的。可以表述为：在自然语言的语料库里，一个单词出现的频率与它在频率表里的排名成反比。）

So if you plot log f versus log r, you should get a straight line with slope −s and intercept log c.

> 因此如果你将 log f 和 log r 进行二维坐标系投点，就应该得到一条直线，斜率是-s，截距是 log c。

Write a program that reads a text from a file, counts word frequencies, and prints one line for each word, in descending order of frequency, with log f and log r.

> 写一个程序，从一个文件中读取文本，统计单词频率，然后每个单词一行来输出，按照词频的降序，同时输出一下 log f 和 log r。

Use the graphing program of your choice to plot the results and check whether they form a straight line. Can you estimate the value of s?

> 选一种投图程序，把结果进行投图，然后检查一下是否为一条直线。
>
> 能否估计一下 s 的值呢？

Solution. To run my solution, you need the plotting module matplotlib. If you installed Anaconda, you already have matplotlib; otherwise you might have to install it.

> 样例代码。要运行刚刚这个代码的话，你需要有投图模块 matplotlib。如果你安装了 Anaconda，就已经有 matplotlib 了；或者你就可能需要安装一下了。
>
> （译者注：matplotlib 的安装方法有很多，比如 pip install matplotlib 或者 easy_install -U matplotlib）

Chapter 14 Files 文件

This chapter introduces the idea of “persistent” programs that keep data in permanent storage, and shows how to use different kinds of permanent storage, like files and databases.

> 本章介绍的内容是『持久的』程序，就是把数据进行永久存储，本章介绍了永久存储的不同种类，比如文件与数据库。

14.1 Persistence 持久

Most of the programs we have seen so far are transient in the sense that they run for a short time and produce some output, but when they end, their data disappears. If you run the program again, it starts with a clean slate.

> 目前为止我们见过的程序大多是很短暂的，它们往往只是运行那么一会，然后产生一些输出，等运行结束了，它们的数据就也都没了。如果你再次运行一个程序，又要从头开始了。

Other programs are persistent: they run for a long time (or all the time); they keep at least some of their data in permanent storage (a hard drive, for example); and if they shut down and restart, they pick up where they left off.

> 另外的一些程序就是持久的：它们运行时间很长（甚至一直在运行）；这些程序还会至少永久保存一部分数据（比如存在硬盘上等等）；然后如果程序关闭了或者重新开始了，也能从之前停留的状态继续工作。

Examples of persistent programs are operating systems, which run pretty much whenever a computer is on, and web servers, which run all the time, waiting for requests to come in on the network.

> 这种有持久性的程序的例子很多，比如操作系统，几乎只要电脑开着，操作系统就要运行；再比如网站服务器，也是要一直开着，等待来自网络上的请求。

One of the simplest ways for programs to maintain their data is by reading and writing text files. We have already seen programs that read text files; in this chapter we will see programs that write them.

> 程序保存数据最简单的方法莫过于读写文本文件。之前我们已经见过一些读取文本文件的程序了；本章中我们会来见识一下写出文本的程序。

An alternative is to store the state of the program in a database. In this chapter I will present a simple database and a module, pickle, that makes it easy to store program data.

> 另一种方法是把程序的状态存到数据库里面。在本章我会演示一种简单的数据库，以及一个 pickle 模块，这个模块大大简化了保存程序数据的过程。

14.2 Reading and writing 读写文件

A text file is a sequence of characters stored on a permanent medium like a hard drive, flash memory, or CD-ROM. We saw how to open and read a file in Section 9.1.

> 文本文件就是一系列的字符串，存储在一个永久介质中，比如硬盘、闪存或者光盘之类的东西里面。
>
> 在9.1的时候我们就看到过如何打开和读取一个文件了。

To write a file, you have to open it with mode 'w' as a second parameter:

> 要写入一个文件，就必须要在打开它的时候用『w』作为第二个参数（译者注：w 就是wirte 的意思了）：

```
>>> fout = open('output.txt', 'w')
>>> fout = open('output.txt', 'w')
```

If the file already exists, opening it in write mode clears out the old data and starts fresh, so be careful! If the file doesn’t exist, a new one is created.

> 如果文件已经存在了，这样用写入的模式来打开，会把旧的文件都清除掉，然后重新写入文件，所以一定要小心！如果文件不存在，程序就会创建一个新的。

open returns a file object that provides methods for working with the file. The write method puts data into the file.

> open 函数会返回一个文件对象，文件对象会提供各种方法来处理文件。write 这个方法就把数据写入到文件中了。

```
>>> line1 = "This here's the wattle,\n"
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
>>> fout.write(line1)
24
```

The return value is the number of characters that were written. The file object keeps track of where it is, so if you call write again, it adds the new data to the end of the file.

> 返回值是已写入字符的数量。文件对象会记录所在位置，所以如果你再次调用write方法，会从文件结尾的地方继续添加新的内容。

```
>>> line2 = "the emblem of our land.\n"
>>> line2 = "the emblem of our land.\n"
>>> fout.write(line2)
>>> fout.write(line2)
24
```

When you are done writing, you should close the file.

> 写完文件之后，你需要用 close 方法来关闭文件。

```
>>> fout.close()
>>> fout.close()
```

If you don't close the file, it gets closed for you when the program ends.

> 如果不 close 这个文件，就要等你的程序运行结束退出的时候，它自己才关闭了。

14.3 Format operator 格式运算符

The argument of write has to be a string, so if we want to put other values in a file, we have to convert them to strings. The easiest way to do that is with str:

> write 方法必须用字符串来做参数，所以如果要把其他类型的值写入文件，就得先转换成字符串才行。最简单的方法就是用 str函数：
>
> > > x = 52 x = 52 fout.write(str(x)) fout.write(str(x)) An alternative is to use the format operator, %. When applied to integers, % is the modulus operator. But when the first operand is a string, % is the format operator.
>
> 另外一个方法就是用格式运算符，也就是百分号%。在用于整数的时候，百分号%是取余数的运算符。但当第一个运算对象是字符串的时候，百分号%就成了格式运算符了。

The first operand is the format string, which contains one or more format sequences, which specify how the second operand is formatted. The result is a string.

> 第一个运算对象也就是说明格式的字符串，包含一个或者更多的格式序列，规定了第二个运算对象的输出格式。返回的结果就是格式化后的字符串了。

For example, the format sequence '%d' means that the second operand should be formatted as a decimal integer:

> 例如，'%d'这个格式序列的意思就是第二个运算对象要被格式化成为一个十进制的整数：

```
>>> camels = 42
>>> camels = 42
>>> '%d' % camels
>>> '%d' % camels
'42'
```

The result is the string '42', which is not to be confused with the integer value 42.

> 你看，经过格式化后，结果就是字符串'42'了，而不是再是整数值42了。

A format sequence can appear anywhere in the string, so you can embed a value in a sentence:

> 这种格式化序列可以放到一个字符串的任何一个位置，这样就可以在一句话里面嵌入一个值了：

```
>>> 'I have spotted %d camels.' % camels
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

If there is more than one format sequence in the string, the second argument has to be a tuple. Each format sequence is matched with an element of the tuple, in order.

> 如果格式化序列有一个以上了，那么第二个参数就必须是一个元组了。每个格式序列对应元组当中的一个元素，次序相同。

The following example uses '%d' to format an integer, '%g' to format a floating-point number, and '%s' to format a string:

> 下面的例子中，用了'%d'来格式化输出整型值，用'%g'来格式化浮点数，'%s'就是给字符串用的了。

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

The number of elements in the tuple has to match the number of format sequences in the string. Also, the types of the elements have to match the format sequences:

> 这就要注意力，如果字符串中格式化序列有多个，那个数一定要和后面的元组中元素数量相等才行。另外格式化序列与元组中元素的类型也必须一样：

```
>>> '%d %d %d' % (1, 2)
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

In the first example, there aren’t enough elements; in the second, the element is the wrong type.

> 第一个例子中，后面元组的元素数量缺一个，所以报错了；第二个例子中，元组里面的元素类型与前面格式不匹配，所以也报错了。

For more information on the format operator, see Here. A more powerful alternative is the string format method, which you can read about at Here.

> 想要对格式运算符进行深入了解，可以点击这里。然后还有一种功能更强大的替代方法，就是用字符串的格式化方法 format，可以点击这里来了解更多细节。

14.4 Filenames and paths 文件名与路径

Files are organized into directories (also called “folders”). Every running program has a “current directory”, which is the default directory for most operations. For example, when you open a file for reading, Python looks for it in the current directory.

> 文件都是按照目录（也叫文件夹）来组织存放的。每一个运行着的程序都有一个当前目录，也就是用来处理绝大多数运算和操作的默认目录。比如当你打开一个文件来读取内容的时候，Python 就从当前目录先来查找这个文件了。

The os module provides functions for working with files and directories (“os” stands for “operating system”). os.getcwd returns the name of the current directory:

> 提供函数来处理文件和目录的是 os 模块（os 就是 operating system即操作系统的缩写）。

```
>>> import os
>>> import os
>>> cwd = os.getcwd()
>>> cwd = os.getcwd()
>>> cwd
>>> cwd
'/home/dinsdale'
```

cwd stands for “current working directory”. The result in this example is/home/dinsdale, which is the home directory of a user named dinsdale.

> cwd 代表的是『current working directory』（即当前工作目录）的缩写。刚刚这个例子中返回的结果是/home/dinsdale，这就是一个名字叫 dinsdale 的人的个人账户所在位置了。

A string like ’/home/dinsdale’ that identifies a file or directory is called a path.

> 像是’/home/dinsdale’这样表示一个文件或者目录的字符串就叫做路径。

A simple filename, like memo.txt is also considered a path, but it is a relative path because it relates to the current directory. If the current directory is/home/dinsdale, the filename memo.txt would refer to /home/dinsdale/memo.txt.

> 一个简单的文件名，比如 memo.txt 也可以被当做路径，但这是相对路径，因为这种路径是指代了文件与当前工作目录的相对位置。如果当前目录是/home/dinsdale，那么 memo.txt 这个文件名指代的就是/home/dinsdale/memo.txt 这个文件了。

A path that begins with / does not depend on the current directory; it is called an absolute path. To find the absolute path to a file, you can use os.path.abspath:

> 用右斜杠/开头的路径不依赖当前目录；这就叫做绝对路径。要找到一个文件的绝对路径，可以用 os.path.abspath：

```
>>> os.path.abspath('memo.txt')
>>> os.path.abspath('memo.txt')
 '/home/dinsdale/memo.txt'
```

os.path provides other functions for working with filenames and paths. For example, os.path.exists checks whether a file or directory exists:

> os.path 提供了其他一些函数，可以处理文件名和路径。比如 os.path.exists 会检查一个文件或者目录是否存在：

```
>>> os.path.exists('memo.txt')
>>> os.path.exists('memo.txt')
True
```

If it exists, os.path.isdir checks whether it's a directory:

> 如果存在，os.path.isdir 可以来检查一下对象是不是一个目录：

```
>>> os.path.isdir('memo.txt')
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('/home/dinsdale')
>>> os.path.isdir('/home/dinsdale')
True
```

Similarly, os.path.isfile checks whether it's a file. os.listdir returns a list of the files (and other directories) in the given directory:

> 同理，os.path.isfile 就可以检查对象是不是一个文件了。
>
> os.listdir 会返回指定目录内的文件（以及次级目录）列表。

```
>>> os.listdir(cwd)
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

To demonstrate these functions, the following example “walks” through a directory, prints the names of all the files, and calls itself recursively on all the directories.

> 为了展示一下这些函数的用法，下面这个例子中，walks 这个函数就遍历了一个目录，然后输出了所有该目录下的文件的名字，并且在该目录下的所有子目录中递归调用自身。

```
def walk(dirname):
    for name in os.listdir(dirname):
        path = os.path.join(dirname, name)
        if os.path.isfile(path):
            print(path)
        else:
            walk(path)
```

os.path.join takes a directory and a file name and joins them into a complete path.

> os.path.join 接收一个目录和一个文件名做参数，然后把它们拼接成一个完整的路径。

The os module provides a function called walk that is similar to this one but more versatile. As an exercise, read the documentation and use it to print the names of the files in a given directory and its subdirectories. You can download my solution from Here.

> os 模块还提供了一个叫 walk 的函数，与上面这个函数很像，功能要更强大一些。做一个练习吧，读一下文档，然后用这个 walk 函数来输出给定目录中的文件名以及子目录的名字。可以从这里下载我的样例代码。

14.5 Catching exceptions 捕获异常

A lot of things can go wrong when you try to read and write files. If you try to open a file that doesn’t exist, you get an IOError:

> 读写文件的时候有很多容易出错的地方。如果你要打开的文件不存在，就会得到一个 IOerror：

```
>>> fin = open('bad_file')
>>> fin = open('bad_file')
IOError: [Errno 2] No such file or directory: 'bad_file'
```

If you don’t have permission to access a file:

> 如果你要读取一个文件却没有权限，就得到一个权限错误permissionError：

```
>>> fout = open('/etc/passwd', 'w')
>>> fout = open('/etc/passwd', 'w')
PermissionError: [Errno 13] Permission denied: '/etc/passwd'
```

And if you try to open a directory for reading, you get

> 如果你把一个目录错当做文件来打开，就会得到下面这种IsADirectoryError错误了：

```
>>> fin = open('/home')
>>> fin = open('/home')
IsADirectoryError: [Errno 21] Is a directory: '/home'
```

To avoid these errors, you could use functions like os.path.exists and os.path.isfile, but it would take a lot of time and code to check all the possibilities (if “Errno 21” is any indication, there are at least 21 things that can go wrong).

> 你可以用像是os.path.exists、os.path.isfile 等等这类的函数来避免上面这些错误，不过这就需要很长时间，还要检查很多代码（比如“Errno 21”就表明有至少21处地方有可能存在错误）。

It is better to go ahead and try—and deal with problems if they happen—which is exactly what the try statement does. The syntax is similar to an if...else statement:

> 所以更好的办法是提前检查，用 try 语句，这种语句就是用来处理异常情况的。其语法形式就跟 if...else 语句是差不多的：

```
try:
    fin = open('bad_file')
except:
    print('Something went wrong.')
```

Python starts by executing the try clause. If all goes well, it skips the except clause and proceeds. If an exception occurs, it jumps out of the try clause and runs the except clause.

> Python 会先执行 try 后面的语句。如果运行正常，就会跳过 except 语句，然后继续运行。如果除了异常，就会跳出 try 语句，然后运行 except 语句中的代码。

Handling an exception with a try statement is called catching an exception. In this example, the except clause prints an error message that is not very helpful. In general, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully.

> 这种用 try 语句来处理异常的方法，就叫异常捕获。上面的例子中，except 语句中的输出信息并没有什么用。一般情况，得到异常之后，你可以选择解决掉这个问题或者再重试一下，或者就以正常状态退出程序了。

14.6 Databases 数据库

A database is a file that is organized for storing data. Many databases are organized like a dictionary in the sense that they map from keys to values. The biggest difference between a database and a dictionary is that the database is on disk (or other permanent storage), so it persists after the program ends.

> 数据库是一个用来管理已存储数据的文件。很多数据库都以类似字典的形式来管理数据，就是从键到键值成对映射。数据库和字典的最大区别就在于数据库是存储在磁盘（或者其他永久性存储设备中），所以程序运行结束退出后，数据库依然存在。
>
> （译者注：这里作者为了便于理解，对数据库的概念进行了极度的简化，实际上数据库的类型、模式、功能等等都与字典有很大不同，比如有关系型数据库和非关系型数据库，还有分布式的和单一文件式的等等。如果有兴趣对数据库进行进一步了解，译者推荐一本书：SQLite Python Tutorial。）

The module dbm provides an interface for creating and updating database files. As an example, I'll create a database that contains captions for image files. Opening a database is similar to opening other files:

> dbm 模块提供了一个创建和更新数据库文件的交互界面。下面这个例子中，我创建了一个数据库，其中的内容是图像文件的标题。
>
> 打开数据库文件就跟打开其他文件差不多：

```
>>> import dbm
>>> import dbm
>>> db = dbm.open('captions', 'c')
>>> db = dbm.open('captions', 'c')
```

The mode 'c' means that the database should be created if it doesn't already exist. The result is a database object that can be used (for most operations) like a dictionary.

> 后面这个 c 是一个模式，意思是如果该数据库不存在就创建一个新的。得到的返回结果就是一个数据库对象了，用起来很多的运算都跟字典很像。

When you create a new item, dbm updates the database file.

> 创建一个新的项的时候，dbm 就会对数据库文件进行更新了。

```
>>> db['cleese.png'] = 'Photo of John Cleese.'
>>> db['cleese.png'] = 'Photo of John Cleese.'
```

When you access one of the items, dbm reads the file:

> 读取里面的某一项的时候，dbm 就读取数据库文件：

```
>>>db['cleese.png']
>>>db['cleese.png']
b'Photo of John Cleese.'
```

The result is a bytes object, which is why it begins with b. A bytes object is similar to a string in many ways. When you get farther into Python, the difference becomes important, but for now we can ignore it.

> 上面的代码返回的结果是一个二进制对象，这也就是开头有个 b 的原因了。二进制对象就跟字符串在很多方面都挺像的。以后对 Python 的学习深入了之后，这种区别就变得很重要了，不过现在还不要紧，咱们就忽略掉。

If you make another assignment to an existing key, dbm replaces the old value:

> 如果对一个已经存在值的键进行赋值，dbm 就会把旧的值替换成新的值：

```
>>> db['cleese.png'] = 'Photo of John Cleese doing a silly walk.'
>>> db['cleese.png'] = 'Photo of John Cleese doing a silly walk.'
>>> db['cleese.png']
>>> db['cleese.png']
b'Photo of John Cleese doing a silly walk.'
```

Some dictionary methods, like keys and items, don't work with database objects. But iteration with a for loop works:

> 字典的一些方法，比如 keys 和 items，是不能用于数据库对象的。但用一个 for 循环来迭代是可以的：

```
for key in db:
    print(key, db[key])
```

As with other files, you should close the database when you are done:

> 然后就同其他文件一样，用完了之后你得用 close 方法关闭数据库：

```
>>> db.close()
>>> db.close()
```

14.7 Pickling Pickle模块

A limitation of dbm is that the keys and values have to be strings or bytes. If you try to use any other type, you get an error.

> dbm 的局限就在于键和键值必须是字符串或者二进制。如果用其他类型数据，就得到错误了。

The pickle module can help. It translates almost any type of object into a string suitable for storage in a database, and then translates strings back into objects.

> 这时候就可以用 pickle 模块了。该模块可以把几乎所有类型的对象翻译成字符串模式，以便存储在数据库中，然后用的时候还可以把字符串再翻译回来。

pickle.dumps takes an object as a parameter and returns a string representation (dumps is short for “dump string”):

> pickle.dumps 接收一个对象做参数，然后返回一个字符串形式的内容翻译（dumps 就是『dump string』的缩写）：

```
>>> import pickle
>>> import pickle
>>> t = [1, 2, 3]
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
>>> pickle.dumps(t)
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

The format isn’t obvious to human readers; it is meant to be easy for pickle to interpret. pickle.loads (“load string”) reconstitutes the object:

> 这种格式让人读起来挺复杂；这种设计能让 pickle 模块解译起来比较容易。pickle.lods("load string")就又会把原来的对象解译出来：

```
>>> t1 = [1, 2, 3]
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)
>>> t2 = pickle.loads(s)
>>> t2
>>> t2
[1, 2, 3]
```

Although the new object has the same value as the old, it is not (in general) the same object:

> 这里要注意了，新的对象与旧的有一样的值，但（通常）并不是同一个对象：

```
>>> t1 == t2
>>> t1 == t2
True
>>> t1 is t2
>>> t1 is t2
False
```

In other words, pickling and then unpickling has the same effect as copying the object.

> 换句话说，就是说 pickle 解译的过程就如同复制了原有对象一样。

You can use pickle to store non-strings in a database. In fact, this combination is so common that it has been encapsulated in a module called shelve.

> 有 pickle了，就可以把非字符串的数据也存到数据库里面了。实际上这种结合方式特别普遍，已经封装到一个叫shelve的模块中了。

14.8 Pipes 管道

Most operating systems provide a command-line interface, also known as a shell. Shells usually provide commands to navigate the file system and launch applications. For example, in Unix you can change directories with cd, display the contents of a directory with ls, and launch a web browser by typing (for example) firefox.

> 大多数操作系统都提供了一个命令行界面，也被称作『shell』。Shell 通常提供了很多基础的命令，能够来搜索文件系统，以及启动应用软件。比如，在 Unix 下面，就可以通过 cd 命令来切换目录，用 ls 命令来显示一个目录下的内容，如果装了火狐浏览器，就可以输入 fireforx 来启动浏览器了。

Any program that you can launch from the shell can also be launched from Python using a pipe object, which represents a running program.

> 在 shell 下能够启动的所有程序，也都可以在 Python 中启动，这要用到一个 pipe 对象，这个直接翻译意思为管道的对象可以理解为 Python 到操作系统的 Shell 进行通信的途径，一个 pipe 对象就代表了一个运行的程序。

For example, the Unix command ls -l normally displays the contents of the current directory in long format. You can launch ls with os.popen:

> 举个例子吧，Unix 的 ls -l 命令通常会用长文件名格式来显示当前目录的内容。在 Python 中就可以用 os.open 来启动它：

```
>>> cmd = 'ls -l'
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
>>> fp = os.popen(cmd)
```

The argument is a string that contains a shell command. The return value is an object that behaves like an open file. You can read the output from the ls process one line at a time with readline or get the whole thing at once with read:

> 参数 cmd 是包含了 shell 命令的一个字符串。返回的结果是一个对象，用起来就像是一个打开了的文件一样。
>
> 可以读取ls 进程的输出，用 readline 的话每次读取一行，用 read 的话就一次性全部读取：

```
>>> res = fp.read()
>>> res = fp.read()
```

When you are done, you close the pipe like a file:

> 用完之后要关闭，这点也跟文件一样：

```
>>> stat = fp.close()
>>> stat = fp.close()
>>> print(stat)
>>> print(stat)
None
```

The return value is the final status of the ls process; None means that it ended normally (with no errors). For example, most Unix systems provide a command called md5sum that reads the contents of a file and computes a “checksum”. You can read about MD5 at Here.

> 返回值是 ls 这个进程的最终状态；None 的意思就是正常退出（没有错误）。
>
> 举个例子，大多数 Unix 系统都提供了一个教唆 md5sum 的函数，会读取一个文件的内容，然后计算一个『checksum』（校验值）。你可以点击这里阅读更多相关内容。

This command provides an efficient way to check whether two files have the same contents. The probability that different contents yield the same checksum is very small (that is, unlikely to happen before the universe collapses).

> 这个命令可以很有效地检查两个文件是否有相同内容。两个不同内容产生同样的校验值的可能性是很小的（实际上在宇宙坍塌之前都没戏）。

You can use a pipe to run md5sum from Python and get the result:

> 你就可以用一个 pipe 来从 Python 启动运行 md5sum，然后获取结果：

```
>>> filename = 'book.tex'
>>> filename = 'book.tex'
>>> cmd = 'md5sum ' + filename
>>> cmd = 'md5sum ' + filename
>>> fp = os.popen(cmd)
>>> fp = os.popen(cmd)
>>> res = fp.read()
>>> res = fp.read()
>>> stat = fp.close()
>>> stat = fp.close()
>>> print(res)
>>> print(res)
1e0033f0ed0656636de0d75144ba32e0  book.tex
>>> print(stat)
>>> print(stat)
None
```

14.9 Writing modules 编写模块

Any file that contains Python code can be imported as a module. For example, suppose you have a file named wc.py with the following code:

> 任何包含 Python 代码的文件都可以作为模块被导入使用。举个例子，假设你有一个名字叫 wc.py 的文件，里面代码如下：

```
def linecount(filename):
    count = 0
    for line in open(filename):
        count += 1
    return count
print(linecount('wc.py'))
```

If you run this program, it reads itself and prints the number of lines in the file, which is 7. You can also import it like this:

> 如果运行这个程序，程序就会读取自己本身，然后输出文件中的行数，也就是7行了。你还可以导入这个模块，如下所示：

```
>>> import wc
>>> import wc
7
```

Now you have a module object wc:

> 现在你就有一个模块对象 wc 了：

```
>>> wc
>>> wc
<module 'wc' from 'wc.py'>
```

The module object provides linecount:

> 该模块提供了数行数的函数linecount：

```
>>> wc.linecount('wc.py')
>>> wc.linecount('wc.py')
7
```

So that’s how you write modules in Python. The only problem with this example is that when you import the module it runs the test code at the bottom. Normally when you import a module, it defines new functions but it doesn’t run them.

> 你看，你就可以这样来为 Python 写模块了。
>
> 当然这个例子中有个小问题，就是导入模块的时候，模块内代码在最后一行对自身进行了测试。
>
> 一般情况你导入一个模块，模块只是定义了新的函数，但不会去主动运行自己内部的函数。

Programs that will be imported as modules often use the following idiom:

> 以模块方式导入使用的程序一般用下面这样的惯用形式：

```
if __name__ == '__main__':
    print(linecount('wc.py'))
```

__name__ is a built-in variable that is set when the program starts. If the program is running as a script, __name__ has the value '__main__'; in that case, the test code runs. Otherwise, if the module is being imported, the test code is skipped.

> __name__ 是一个内置变量，当程序开始运行的时候被设置。如果程序是作为脚本来运行的，__name__ 的值就是'__main__'；这样的话，if条件满足，测试代码就会运行。而如果该代码被用作模块导入了，if 条件不满足，测试的代码就不会运行了。

As an exercise, type this example into a file named wc.py and run it as a script. Then run the Python interpreter and import wc. What is the value of __name__when the module is being imported?

> 做个联系吧，把上面的例子输入到一个名为 wc.py 的文件中，然后作为脚本运行。然后再运行 Python 解释器，然后导入 wc 作为模块。看看作为模块导入的时候__name__ 的值是什么？

Warning: If you import a module that has already been imported, Python does nothing. It does not re-read the file, even if it has changed.

> 警告：如果你导入了一个已经导入过的模块，Python 是不会有任何提示的。Python 并不会重新读取模块文件，即便该文件又被修改过也是如此。

If you want to reload a module, you can use the built-in function reload, but it can be tricky, so the safest thing to do is restart the interpreter and then import the module again.

> 所以如果你想要重新加在一个模块，你可以用内置函数 reload，但这个也不太靠谱，所以最靠谱的办法莫过于重启解释器，然后再次导入该模块。

14.10 Debugging 调试

When you are reading and writing files, you might run into problems with whitespace. These errors can be hard to debug because spaces, tabs and newlines are normally invisible:

> 读写文件的时候，你可能会碰到空格导致的问题。这些问题很难解决，因为空格、跳表以及换行，平常就难以用眼睛看出来：

```
>>> s = '1 2\t 3\n 4'
>>> s = '1 2\t 3\n 4'
>>> print(s)
>>> print(s)
1 2  3
4
```

The built-in function repr can help. It takes any object as an argument and returns a string representation of the object. For strings, it represents whitespace characters with backslash sequences:

> 这时候就可以用内置函数 repr 来帮忙。它接收任意对象作为参数，然后返回一个该对象的字符串表示。对于字符串，该函数可以把空格字符转成反斜杠序列：

```
>>> print(repr(s))
>>> print(repr(s))
'1 2\t 3\n 4'
```

This can be helpful for debugging.

> 该函数的功能对调试来说很有帮助。

One other problem you might run into is that different systems use different characters to indicate the end of a line. Some systems use a newline, represented \n. Others use a return character, represented \r. Some use both. If you move files between different systems, these inconsistencies can cause problems.

> 另外一个问题就是不同操作系统可能用不同字符表示行尾。
>
> 有的用一个换行符，也就是\n。有的用一个返回字符，也就是\r。有的两个都亏。如果你把文件在不同操作系统只见移动，这种不兼容性就可能导致问题了。

For most systems, there are applications to convert from one format to another. You can find them (and read more about this issue) at Here. Or, of course, you could write one yourself.

> 对大多数操作系统，都有一些应用软件来进行格式转换。你可以在这里查找一下（并且阅读关于该问题的更多细节）。当然，你也可以自己写一个转换工具了。
>
> （译者注：译者这里也鼓励大家，一般的小工具，自己有时间有精力的话完全可以尝试着自己写一写，对自己是个磨练，也有利于对语言进行进一步的熟悉。这里再推荐一本书：Automate the Boring Stuff with，作者是 Al Sweigart。该书里面提到了很多常用的任务用 Python 来实现。）

14.11 Glossary 术语列表

persistent: Pertaining to a program that runs indefinitely and keeps at least some of its data in permanent storage.

> 持久性：指一个程序可以随时运行，然后可以存储一部分数据到永久介质中。

format operator: An operator, %, that takes a format string and a tuple and generates a string that includes the elements of the tuple formatted as specified by the format string.

> 格式运算符：%运算符，处理字符串和元组，然后生成一个包含元组中元素的字符串，根据给定的格式字符串进行格式化。

format string: A string, used with the format operator, that contains format sequences.

> 格式字符串：用于格式运算符的一个字符串，内含格式序列。

format sequence: A sequence of characters in a format string, like %d, that specifies how a value should be formatted.

> 格式序列：格式字符串内的一串字符，比如%d，规定了一个值如何进行格式化。

text file: A sequence of characters stored in permanent storage like a hard drive.

> 文本文件：磁盘中永久存储的一个文件，内容为一系列的字符。

directory: A named collection of files, also called a folder.

> 目录：有名字的文件集合，也叫做文件夹。

path: A string that identifies a file.

> 路径：指向某个文件的字符串。

relative path: A path that starts from the current directory.

> 相对路径：从当前目录开始，到目标文件的路径。

absolute path: A path that starts from the topmost directory in the file system.

> 绝对路径：从文件系统最底层的根目录开始，到目标文件的路径。

catch: To prevent an exception from terminating a program using the try and except statements.

> 抛出异常：为了避免意外错误中止程序，使用 try 和 except 语句来处理异常。

database: A file whose contents are organized like a dictionary with keys that correspond to values.

> 数据库：一个文件，全部内容以类似字典的方式来组织，为键与对应的键值。

bytes object: An object similar to a string.

> 二进制对象：暂时就当作是根字符串差不多的对象就可以了。

shell: A program that allows users to type commands and then executes them by starting other programs.

> shell：一个程序，允许用户与操作系统进行交互，可以输入命令，然后启动一些其他程序来执行。

pipe object: An object that represents a running program, allowing a Python program to run commands and read the results.

> 管道对象：代表了一个正在运行的程序的对象，允许一个 Python 程序运行命令并读取运行结果。

14.12 Exercises 练习

Exercise 1 练习1

Write a function called sed that takes as arguments a pattern string, a replacement string, and two filenames; it should read the first file and write the contents into the second file (creating it if necessary). If the pattern string appears anywhere in the file, it should be replaced with the replacement string.

> 写一个函数，名为 sed，接收一个目标字符串，一个替换字符串，然后两个文件名；读取第一个文件，然后把内容写入到第二个文件中，如果第二个文件不存在，就创建一个。如果目标字符串在文件中出现了，就用替换字符串把它替换掉。

If an error occurs while opening, reading, writing or closing files, your program should catch the exception, print an error message, and exit. Solution.

> 如果在打开、读取、写入或者关闭文件的时候发生了错误了，你的程序应该要捕获异常，然后输出错误信息，然后再退出。样例代码。

Exercise 2 练习2

If you download my solution to Exercise 2 from Here, you'll see that it creates a dictionary that maps from a sorted string of letters to the list of words that can be spelled with those letters. For example, 'opst' maps to the list ['opts', 'post', 'pots', 'spot', 'stop', 'tops'].

> 如果你从 这里下载了我的样例代码，你会发现该程序创建了一个字典，建立了从一个有序字母字符串到一个单词列表的映射，列表中的单词可以由这些字母拼成。例如'opst'就映射到了列表 ['opts', 'post', 'pots', 'spot', 'stop', 'tops'].

Write a module that imports anagram_sets and provides two new functions:store_anagrams should store the anagram dictionary in a “shelf”;read_anagrams should look up a word and return a list of its anagrams. Solution.

> 写一个模块，导入 anagram_sets 然后提供两个函数：store_anagrams 可以把相同字母异序词词典存储到一个『shelf』；read_anagrams 可以查找一个词，返回一个由其 相同字母异序词 组成的列表。
>
> 样例代码。

Exercise 3 练习3

In a large collection of MP3 files, there may be more than one copy of the same song, stored in different directories or with different file names. The goal of this exercise is to search for duplicates.

> 现在有很多 MP3文件的一个大集合里面，一定有很多同一首歌重复了，然后存在不同的目录或者保存的名字不同。本次练习的目的就是要找到这些重复的内容。

1. Write a program that searches a directory and all of its subdirectories, recursively, and returns a list of complete paths for all files with a given suffix (like .mp3). Hint: os.path provides several useful functions for manipulating file and path names.

> 首先写一个程序，搜索一个目录并且递归搜索所有子目录，然后返回一个全部给定后缀（比如.mp3）的文件的路径。提示：os.path 提供了一些函数，能用来处理文件和路径名称。

1. To recognize duplicates, you can use md5sum to compute a “checksum” for each files. If two files have the same checksum, they probably have the same contents.

> 要识别重复文件，要用到 md5sum 函数来对每一个文件计算一个『校验值』。如果两个文件校验值相同，那很可能就是有同样的内容了。

1. To double-check, you can use the Unix command diff.

> 为了保险起见，再用 Unix 的 diff 命令来检查一下。

Solution.

> 样例代码。

备注1 popen is deprecated now, which means we are supposed to stop using it and start using the subprocess module. But for simple cases, I find subprocess more complicated than necessary. So I am going to keep using popen until they take it away.

> 注意，popen 已经不被支持了，这就意味着咱们不应该再用它了，然后要用新的 subprocess 模块。不过为了让案例更简单明了，还是用了 popen，引起我发现 subprocess 过于复杂，而且也没太大必要。所以我就打算一直用着 popen，直到这个方法被废弃移除不能使用了再说了。

Chapter 15 Classes and objects 类和对象

At this point you know how to use functions to organize code and built-in types to organize data. The next step is to learn “object-oriented programming”, which uses programmer-defined types to organize both code and data. Object-oriented programming is a big topic; it will take a few chapters to get there. Code examples from this chapter are available from Here; solutions to the exercises are available from Here.

> 到目前位置，你应该已经知道如何用函数来整理代码，以及用内置类型来组织数据了。接下来的一步就是要学习『面向对象编程』了，这种编程方法中，用户可以自定义类型来同时对代码和数据进行整理。面向对象编程是一个很大的题目；要有好几章才能讲出个大概。
>
> 本章的样例代码可以在这里来下载，练习题对应的样例代码可以在这里下载。

15.1 Programmer-defined types 用户自定义类型

We have used many of Python’s built-in types; now we are going to define a new type. As an example, we will create a type called Point that represents a point in two-dimensional space.

> 我们已经用过很多 Python 的内置类型了；现在我们就要来定义一个新的类型了。作为样例，我们会创建一个叫 Point 的类，用于表示一个二维空间中的点。

In mathematical notation, points are often written in parentheses with a comma separating the coordinates. For example, (0,0) represents the origin, and (x,y) represents the point x units to the right and y units up from the origin.

> 数学符号上对点的表述一般是一个括号内有两个坐标，坐标用逗号分隔开。比如，（0，0）就表示为原点，（x,y）就表示了该点从原点向右偏移 x，向上偏移 y。

There are several ways we might represent points in Python:

> 我们可以用好几种方法来在 Python 中表示一个点：

• We could store the coordinates separately in two variables, x and y.

> 我们可以把坐标存储成两个单独的值，x 和 y。

• We could store the coordinates as elements in a list or tuple.

> 还可以把坐标存储成列表或者元组的元素。

• We could create a new type to represent points as objects.

> 还可以创建一个新的类型来用对象表示点。

Creating a new type is more complicated than the other options, but it has advantages that will be apparent soon.

> 创建新的类型要比其他方法更复杂一点，不过也有一些优势，等会我们就会发现了。

A programmer-defined type is also called a class. A class definition looks like this:

> 用户自定义的类型也被叫做一个类。一个类的定义大概是如下所示的样子：

```
class Point:
"""Represents a point in 2-D space."""
```

The header indicates that the new class is called Point. The body is a docstring that explains what the class is for. You can define variables and methods inside a class definition, but we will get back to that later.

> 头部代码的意思是表示新建的类名字叫 Point。然后类的体内有一个文档字符串，解释类的用途。在类的定义内部可以定义各种变量和方法，等会再来详细学习一下这些内容哈。

Defining a class named Point creates a class object.

> 声明一个名为 Point 的类，就可以创建该类的一个对象。

```
>>> Point
>>> Point
<class '__main__.Point'>
```

Because Point is defined at the top level, its “full name” is __main__.Point.

> 因为 Point 是在顶层位置定义的，所以全名就是__main__.Point。

The class object is like a factory for creating objects. To create a Point, you call Point as if it were a function.

> 类的对象就像是一个创建对象的工厂。要创建一个 Point，就可以像调用函数一样调用 Point。

```
>>> blank = Point()
>>> blank = Point()
>>> blank
>>> blank
<__main__.Point object at 0xb7e9d3ac>
```

The return value is a reference to a Point object, which we assign to blank. Creating a new object is called instantiation, and the object is an instance of the class.

> 返回值是到一个 Point 对象的引用，刚刚赋值为空白了。
>
> 创建一个新的对象也叫做实例化，这个对象就是类的一个实例。

When you print an instance, Python tells you what class it belongs to and where it is stored in memory (the prefix 0x means that the following number is in hexadecimal).

> 用 Print 输出一个实例的时候，Python 会告诉你该实例所属的类，以及在内存中存储的位置（前缀为0x 意味着下面的这些数值是十六进制的。）

Every object is an instance of some class, so “object” and “instance” are interchangeable. But in this chapter I use “instance” to indicate that I am talking about a programmer-defined type.

> 每一个对象都是某一个类的一个实例，所以『对象』和『实例』可以互换来使用。不过本章我还是都使用『实例』这个词，这样就更能体现出咱们在谈论的是用户定义的类型。

15.2 Attributes 属性

You can assign values to an instance using dot notation:

> 用点号可以给实例进行赋值：

```
>>> blank.x = 3.0
>>> blank.x = 3.0
>>> blank.y = 4.0
>>> blank.y = 4.0
```

This syntax is similar to the syntax for selecting a variable from a module, such as math.pi or string.whitespace. In this case, though, we are assigning values to named elements of an object. These elements are called attributes.

> 这一语法形式就和从模块中选取变量的语法是相似的，比如 math.pi 或者 string.whitespace。然而在本章这种情况下，我们用点号实现的是对一个对象中某些特定名称的元素进行赋值。这些元素也叫做属性。

As a noun, “AT-trib-ute” is pronounced with emphasis on the first syllable, as opposed to “a-TRIB-ute”, which is a verb.

> 『Attribute』作为名词的发音要把重音放在第一个音节，而做动词的时候是重音放到第二音节。

The following diagram shows the result of these assignments. A state diagram that shows an object and its attributes is called an object diagram; see Figure 15.1.

> 下面的图表展示了上面这些赋值的结果。用于展示一个类及其属性的状态图也叫做类图；比如图15.1就是一例。

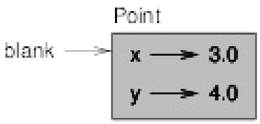


Figure 15.1: Object diagram.

The variable blank refers to a Point object, which contains two attributes. Each attribute refers to a floating-point number.

> 变量 blank 指代的是一个 Point 对象，该对象包含两个属性。每个属性都指代了一个浮点数。

You can read the value of an attribute using the same syntax:

> 读取属性值可以用如下这样的语法：

```
>>> blank.y
>>> blank.y
4.0
>>> x = blank.x
>>> x = blank.x
>>> x
>>> x
3.0
```

The expression blank.x means, “Go to the object blank refers to and get the value of x.” In the example, we assign that value to a variable named x. There is no conflict between the variable x and the attribute x.

> 这里的表达式 blank.x 的意思是，『到 blank 所指代的对象中，读取 x 的值。』在这个例子中，我们把这个值赋值给一个名为 x 的变量。这里的变量 x 和类的属性x 并不冲突。

You can use dot notation as part of any expression. For example:

> 点号可以随意在任意表达式中使用。比如下面这个例子：

```
>>> '(%g, %g)' % (blank.x, blank.y)
>>> '(%g, %g)' % (blank.x, blank.y)
'(3.0, 4.0)'
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> distance
>>> distance
5.0
```

You can pass an instance as an argument in the usual way. For example:

> 你还可以把实例作为一个参数来使用。比如下面这样：

```
def print_point(p):
    print('(%g, %g)' % (p.x, p.y))
```

print_point takes a point as an argument and displays it in mathematical notation. To invoke it, you can pass blank as an argument:

> print_point 这个函数就接收了一个点作为参数，然后显示点的数值位置。你可以把刚刚那个 blank 作为参数传过去来试试：

```
>>> print_point(blank)
>>> print_point(blank)
(3.0, 4.0)
```

Inside the function, p is an alias for blank, so if the function modifies p, blank changes.

> 在函数内部，p 是blank 的一个别名，所以如果函数内部对 p 进行了修改，blank 也会发生相应的改变。

As an exercise, write a function called distance_between_points that takes two Points as arguments and returns the distance between them.

> 做个练习，写一个名为 distance_between_points 的函数，接收两个点作为参数，然后返回两点之间的距离。

15.3 Rectangles 矩形

Sometimes it is obvious what the attributes of an object should be, but other times you have to make decisions. For example, imagine you are designing a class to represent rectangles. What attributes would you use to specify the location and size of a rectangle? You can ignore angle; to keep things simple, assume that the rectangle is either vertical or horizontal.

> 有时候一个类中的属性应该如何设置是很明显的，不过有的时候就得好好考虑一下了。比如，假设你要设计一个表示矩形的类。你要用什么样的属性来确定一个矩形的位置和大小呢？可以忽略角度；来让情况更简单一些，就只考虑矩形是横向的或者纵向的。

There are at least two possibilities:

> 至少有两种方案备选：

• You could specify one corner of the rectangle (or the center), the width, and the height.

> 确定矩形的一个顶点（或者中心）所在位置，还有宽度和高度。

• You could specify two opposing corners.

> 确定对角线上的两个顶点所在位置。

At this point it is hard to say whether either is better than the other, so we'll implement the first one, just as an example.

> 现在还很难说这两者哪一个更好，那么咱们先用第一个方案来做个例子。

Here is the class definition:

> 下面就是类的定义：

```
class Rectangle:
"""Represents a rectangle.
attributes: width, height, corner.
"""
```

The docstring lists the attributes: width and height are numbers; corner is a Point object that specifies the lower-left corner.

> 文档字符串中列出了属性：width 和 height 是数值；corner 是一个点对象，用来表示左下角顶点。

To represent a rectangle, you have to instantiate a Rectangle object and assign values to the attributes:

> 要表示一个矩形，必须初始化一个矩形对象，然后对其属性进行赋值：

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

The expression box.corner.x means, "Go to the object box refers to and select the attribute named corner; then go to that object and select the attribute named x."

> 表达式 box.corner.x 的意思是，『到 box 指代的对象中，选择名为 corner 的属性；然后到这个点对象中，选取名为 x 的属性值。』

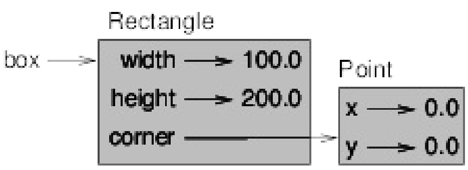


Figure 15.2: Object diagram.

Figure 15.2 shows the state of this object. An object that is an attribute of another object is embedded.

> 图15.2展示了这个对象的状态图。一个类去作为另外一个类的属性，就叫做嵌入。

15.4 Instances as return values 多个实例作返回值

Functions can return instances. For example, find_center takes a Rectangle as an argument and returns a Point that contains the coordinates of the center of the Rectangle:

> 函数亏返回实例。比如 find_center 就接收一个 Rectangle （矩阵）对象作为参数，然后以一个Point（点）对象的形式返回矩形中心位置的坐标所在点：

```
def find_center(rect):
    p = Point()
    p.x = rect.corner.x + rect.width/2
    p.y = rect.corner.y + rect.height/2
    return p
```

Here is an example that passes box as an argument and assigns the resulting Point to center:

> 下面这个例子中，box 作为一个参数传递给了 find_center 函数，然后结果赋值给了点 center：

```
>>> center = find_center(box)
>>> center = find_center(box)
>>> print_point(center)
>>> print_point(center)
(50, 100)
```

15.5 Objects are mutable 对象可以修改

You can change the state of an object by making an assignment to one of its attributes. For example, to change the size of a rectangle without changing its position, you can modify the values of width and height:

> 通过对一个对象的属性进行赋值就可以修改该对象的状态了。比如，要改变一个举行的大小而不改变位置，就可以只修改宽度和高度，如下所示：

```
box.width = box.width + 50
box.height = box.height + 100
```

You can also write functions that modify objects. For example, grow_rectangle takes a Rectangle object and two numbers, dwidth and dheight, and adds the numbers to the width and height of the rectangle:

> 你还可以写专门的函数来修改对象。比如grow_rectangle这个函数就接收一个矩形对象和dwidth 与 dheight两个数值，然后把这两个数值加到矩形的宽度和高度值上。

```
def grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

Here is an example that demonstrates the effect:

> 下面的例子展示了具体的效果：

```
>>> box.width, box.height
>>> box.width, box.height
(150.0, 300.0)
>>> grow_rectangle(box, 50, 100)
>>> grow_rectangle(box, 50, 100)
>>> box.width, box.height
>>> box.width, box.height
(200.0, 400.0)
```

Inside the function, rect is an alias for box, so when the function modifies rect,box changes.

> 在函数的内部，rect 是 box 的一个别名，所以当函数修改了 rect 的时候，box 就得到了相应的修改。

As an exercise, write a function named move_rectangle that takes a Rectangle and two numbers named dx and dy. It should change the location of the rectangle by adding dx to the x coordinate of corner and adding dy to the y coordinate of corner.

> 做个练习，写一个名为 move_rectangle 的函数，接收一个矩形和dx 与 dy 两个数值。函数要改变矩形所在位置，具体的改变方法为对左下角顶点坐标的 x 和 y 分别加上 dx 和 dy 的值。

15.6 Copying 复制

Aliasing can make a program difficult to read because changes in one place might have unexpected effects in another place. It is hard to keep track of all the variables that might refer to a given object.

> 别名有可能让程序读起来有困难，因为在一个位置做出的修改有可能导致另外一个位置发生不可预知的情况。这样也很难去追踪指向一个对象的所有变量。

Copying an object is often an alternative to aliasing. The copy module contains a function called copy that can duplicate any object:

> 所以就可以不用别名，而用复制对象的方法。copy 模块包含了一个名叫 copy 的函数，可以复制任意对象：

```
>>> p1 = Point()
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.x = 3.0
>>> p1.y = 4.0
>>> p1.y = 4.0
>>> import copy
>>> import copy
>>> p2 = copy.copy(p1)
>>> p2 = copy.copy(p1)
```

p1 and p2 contain the same data, but they are not the same Point.

> p1和 p2包含的数据是相同的，但并不是同一个点对象。

```
>>> print_point(p1)
>>> print_point(p1)
(3, 4)
>>> print_point(p2)
>>> print_point(p2)
(3, 4)
>>> p1 is p2
>>> p1 is p2
False
>>> p1 == p2
>>> p1 == p2
False
```

The is operator indicates that p1 and p2 are not the same object, which is what we expected. But you might have expected == to yield True because these points contain the same data. In that case, you will be disappointed to learn that for instances, the default behavior of the == operator is the same as the is operator; it checks object identity, not object equivalence. That's because for programmer-defined types, Python doesn't know what should be considered equivalent. At least, not yet.

> is 运算符表明 p1和 p2不是同一个对象，这就是我们所预料的。但你可能本想着是==运算符应该得到的是 True 因为这两个点包含的数据是一样的。这样的话你就会很失望地发现对于实例来说，==运算符的默认行为就跟 is 运算符是一样的；它也还是检查对象的身份，而不是对象的相等性。这是因为你用的是用户自定义的类型，Python 不值得如何去衡量是否相等。至少是现在还不能。
>
> （译者注：==运算符的实现需要运算符重载，也就是多态的一种，来实现，也就是对用户自定义类型，需要用户自定义运算符，而不能简单地继续用内置运算符。因为自定义类型的运算是 Python 没法确定的，得用户自己来确定。）

If you use copy.copy to duplicate a Rectangle, you will find that it copies the Rectangle object but not the embedded Point.

> 如果你用 copy.copy 复制了一个矩形，你会发现该函数复制了矩形对象，但没有复制内嵌的点对象。

```
>>> box2 = copy.copy(box)
>>> box2 = copy.copy(box)
>>> box2 is box
>>> box2 is box
False
>>> box2.corner is box.corner
>>> box2.corner is box.corner
True
```

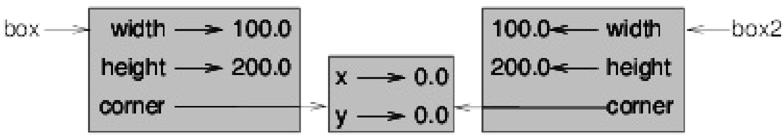


Figure 15.3: Object diagram.

Figure 15.3 shows what the object diagram looks like. This operation is called a shallow copy because it copies the object and any references it contains, but not the embedded objects.

> 图15.3展示了此时的类图的情况。这种运算叫做浅复制，因为复制了对象与对象内包含的所有引用，但不复制内嵌的对象。

For most applications, this is not what you want. In this example, invoking grow_rectangle on one of the Rectangles would not affect the other, but invoking move_rectangle on either would affect both! This behavior is confusing and error-prone.

> 对于大多数应用来说，这并不是你的本来目的。在本节的样例中，对复制过的一个矩形进行 grow_rrectangle 函数运算，并不会影响另外一个，但使用 move_rectangle 就会对两个都有影响！这种行为就很让人疑惑，也容易带来错误。

Fortunately, the copy module provides a method named deepcopy that copies not only the object but also the objects it refers to, and the objects they refer to, and so on. You will not be surprised to learn that this operation is called a deep copy.

> 所幸的是 copy 模块还提供了一个名为 deepcopy （深复制）的方法，这样就能把内嵌的对象也复制了。你肯定不会奇怪了，这种运算就叫深复制了。

```
>>> box3 = copy.deepcopy(box)
>>> box3 = copy.deepcopy(box)
>>> box3 is box
>>> box3 is box
False
>>> box3.corner is box.corner
>>> box3.corner is box.corner
False
```

box3 and box are completely separate objects. As an exercise, write a version of move_rectangle that creates and returns a new Rectangle instead of modifying the old one.

> box3和 box 就是完全隔绝开，没有公用内嵌对象，彻底不会相互干扰的两个对象了。
>
> 做个联系吧，写一个新版本的 move_rectangle，创建并返回一个新的矩形，而不是修改旧有的矩形。

15.7 Debugging 调试

When you start working with objects, you are likely to encounter some new exceptions. If you try to access an attribute that doesn't exist, you get an AttributeError:

> 当你开始使用对象的时候，你就容易遇到一些新的异常。如果你试图读取一个不存在的属性，就会得到一个属性错误AttributeError：

```
>>> p = Point()
>>> p = Point()
>>> p.x = 3
>>> p.x = 3
>>> p.y = 4
>>> p.y = 4
>>> p.z
>>> p.z
AttributeError: Point instance has no attribute 'z'
```

If you are not sure what type an object is, you can ask:

> 如果不确定一个对象是什么类型，可以『问』一下：

```
>>> type(p)
>>> type(p)
<class '__main__.Point'>
```

You can also use isinstance to check whether an object is an instance of a class:

> 还可以用 isinstance 函数来检查一下一个对象是否为某一个类的实例：

```
>>> isinstance(p, Point)
>>> isinstance(p, Point)
True
```

If you are not sure whether an object has a particular attribute, you can use the built-in function hasattr:

> 如果不确定某一对象是否有一个特定的属性，可以用内置函数 hasattr：

```
>>> hasattr(p, 'x')
>>> hasattr(p, 'x')
True
>>> hasattr(p, 'z')
>>> hasattr(p, 'z')
False
```

The first argument can be any object; the second argument is a string that contains the name of the attribute.

> hasattr 的第一个参数可以是任意一个对象；第二个参数是一个字符串，就是要判断是否存在的属性名字。

You can also use a try statement to see if the object has the attributes you need:

> 用 try 语句也可以试验一个对象是否有你需要的属性：

```
try:
    x = p.x
except AttributeError:
    x = 0
```

This approach can make it easier to write functions that work with different types; more on that topic is coming up in Section 17.9.

> 这样写一些处理不同类型变量的函数就更容易了；关于这一话题的更多内容会在17.9中展开。

15.8 Glossary 术语列表

class: A programmer-defined type. A class definition creates a new class object.

> 类：用户定义的类型。一个类的声明建立了一个新的类的对象。

class object: An object that contains information about a programmer-defined type. The class object can be used to create instances of the type.

> 类的对象：包含了用户自定义类型相关信息的一个对象。可以用于创建类的一个实例。

instance: An object that belongs to a class.

> 实例：术语某一个类的一个对象。

instantiate: To create a new object.

> 实例化：创建一个新的对象。

attribute: One of the named values associated with an object.

> 属性：一个对象内附属的数值的名字。

embedded object: An object that is stored as an attribute of another object.

> 内嵌对象：一个对象作为属性存储在另一个对象内。

shallow copy: To copy the contents of an object, including any references to embedded objects; implemented by the copy function in the copy module.

> 浅复制：复制一个对象中除了内嵌对象之外的所有引用；通过 copy 模块的 copy 函数来实现。

deep copy: To copy the contents of an object as well as any embedded objects, and any objects embedded in them, and so on; implemented by the deepcopy function in the copy module.

> 深复制：复制一个对象的所有内容，包括内嵌对象，以及内嵌对象中的所有内嵌对象等等；通过 copy 模块的 deepcopy 函数来实现。

object diagram: A diagram that shows objects, their attributes, and the values of the attributes.

> 类图：一种图解，用于展示类与类中的属性以及属性的值。

15.9 Exercises 练习

Exercise 1 练习1

Write a definition for a class named Circle with attributes center and radius, where center is a Point object and radius is a number.

> 写一个名为 Circle 的类的定义，属性为圆心center和半径radius，center 是一个点对象，半径是一个数值。

Instantiate a Circle object that represents a circle with its center at (150, 100) and radius 75.

> 实例化一个 Circle 的对象，表示一个圆，圆心在（150，100），半径为75。

Write a function named point_in_circle that takes a Circle and a Point and returns True if the Point lies in or on the boundary of the circle.

> 写一个名为 point_in_circle 的函数，接收一个 Circle 和一个 Point 对象作为参数，如果点在圆内或者圆的线上就返回True。

Write a function named rect_in_circle that takes a Circle and a Rectangle and returns True if the Rectangle lies entirely in or on the boundary of the circle.

> 写一个名为 rect_in_circle 的函数，接收一个 Circle 和一个 Rectangle 对象作为参数，如果矩形的边都内含或者内切在圆内，就返回 True。

Write a function named rect_circle_overlap that takes a Circle and a Rectangle and returns True if any of the corners of the Rectangle fall inside the circle. Or as a more challenging version, return True if any part of the Rectangle falls inside the circle.

> 写一个名为 rect_circle_overlap 的函数，接收一个 Circle 和一个 Rectangle 对象作为参数，如果矩形任意一个顶点在圆内就返回 True。或者写个更有挑战性的版本，如果矩形有任意部分包含在圆内就返回 True。

Solution.

> 样例代码。

Exercise 2 练习2

Write a function called draw_rect that takes a Turtle object and a Rectangle and uses the Turtle to draw the Rectangle. See Chapter 4 for examples using Turtle objects.

> 写一个名为 draw_rect的函数，接收一个 Turtle 对象和一个 Rectangle 对象作为参数，用 Turtle 画出这个矩形。可以参考一下第四章对 Turtle 对象使用的样例。

Write a function called draw_circle that takes a Turtle and a Circle and draws the Circle. Solution.

> 写一个名为 draw_circle 的函数，接收一个 Turtle 对象和一个 Circle 对象，画个圆这次。
>
> 样例代码。

Chapter 16 Classes and functions 类和函数

Now that we know how to create new types, the next step is to write functions that take programmer-defined objects as parameters and return them as results. In this chapter I also present "functional programming style" and two new program development plans.

> 现在我们已经知道如何创建新类型了，下一步就要写一些函数了，这些函数用自定义类型做参数和返回值。在本章中还提供了一种函数式编程的模式，以及两种新的程序开发规划方式。

Code examples from this chapter are available from Here. Solutions to the exercises are at Here.

> 本章的样例代码可以在这里下载。然后练习题的样例代码可以在这里下载到。

16.1 Time 时间

As another example of a programmer-defined type, we'll define a class called Time that records the time of day. The class definition looks like this:

> 下面又是一个自定义类型的例子，这次咱们定义一个叫做 Time 的类，记录下当日的时间。
>
> 类的定义是如下这样：

```
class Time:
"""Represents the time of day.
attributes: hour, minute, second     """
```

We can create a new Time object and assign attributes for hours, minutes, and seconds:

> 我们可以建立一个新的 Time 对象，然后对时分秒分别进行赋值：

```
time = Time()
time.hour = 11
time.minute = 59
time.second = 30
```

The state diagram for the Time object looks like Figure 16.1.

> 这个 Time 对象的状态图如图16.1所示。

As an exercise, write a function called print_time that takes a Time object and prints it in the form hour:minute:second. Hint: the format sequence '%.2d' prints an integer using at least two digits, including a leading zero if necessary.

> 下面做个练习，写一个名为print_time 的函数，接收一个 Time 对象，然后以时：分：秒的格式来输出。提示：格式序列'%.2d'就会用两位来输出一个整数，第一位可以为0。

Write a boolean function called is_after that takes two Time objects, t1 andt2, and returns True if t1 follows t2 chronologically and False otherwise. Challenge: don't use an if statement.

> 写一个布尔函数，名为 is_after，接收两个 Time 对象，分别为 t1和 t2，然后如果 t1在时间上位于 t2的后面，就返回真，否则返回假。难度提高一下：不要用 if 语句，看你能搞定不。

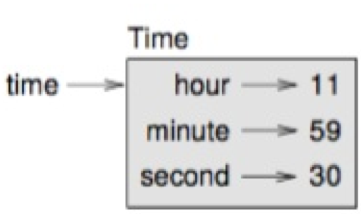


Figure 16.1: Object diagram.

16.2 Pure functions 纯函数

In the next few sections, we'll write two functions that add time values. They demonstrate two kinds of functions: pure functions and modifiers. They also demonstrate a development plan I'll call prototype and patch, which is a way of tackling a complex problem by starting with a simple prototype and incrementally dealing with the complications.

> 后面的这些章节中，我们要写两个函数来对 time 进行加法操作。这两个函数展示了两种函数类型：纯函数和修改器。写这两个函数的过程中，也体现了我即将讲到的一种新的开发模式：原型和补丁模式，这种方法就是在处理复杂问题的时候，先从简单的原型开始，然后逐渐解决复杂的内容。

Here is a simple prototype of add_time:

> 下面这段代码就是 add_time 函数的一个原型：

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second
    return sum
```

The function creates a new Time object, initializes its attributes, and returns a reference to the new object. This is called a pure function because it does not modify any of the objects passed to it as arguments and it has no effect, like displaying a value or getting user input, other than returning a value.

> 这个函数新建了一个新的 Time 对象，初始化了所有的值，然后返回了一个对新对象的引用。这种函数叫纯函数，因为这种函数并不修改传来做参数的对象，也没有什么效果，比如显示值啊或者让用户输入啊等等，而只是返回一个值而已。

To test this function, I'll create two Time objects: start contains the start time of a movie, like Monty Python and the Holy Grail, and duration contains the run time of the movie, which is one hour 35 minutes. add_time figures out when the movie will be done.

> 下面就来测试一下这个函数，我将建立两个 Time 对象，start 包含了一个电影的开始时间，比如《巨蟒与圣杯》（译者注：1975年喜剧电影。Python的创造者Guido van Rossum特别喜欢这个喜剧团体：巨蟒飞行马戏团（Monty Python's Flying Circus ），所以命名为 Python。），然后 duration（汉译就是持续时间）包含了该电影的时长，《巨蟒与圣杯》这部电影是一小时三十五分钟。add_time 函数就会算出电影结束的时间。

```
>>> start = Time()
>>> start = Time()
>>> start.hour = 9
>>> start.hour = 9
>>> start.minute = 45
>>> start.minute = 45
>>> start.second =  0
>>> start.second =  0
>>> duration = Time()
>>> duration = Time()
>>> duration.hour = 1
>>> duration.hour = 1
>>> duration.minute = 35
>>> duration.minute = 35
>>> duration.second = 0
>>> duration.second = 0
>>> done = add_time(start, duration)
>>> done = add_time(start, duration)
>>> print_time(done)
>>> print_time(done)
10:80:00
```

The result, 10:80:00 might not be what you were hoping for. The problem is that this function does not deal with cases where the number of seconds or minutes adds up to more than sixty. When that happens, we have to "carry" the extra seconds into the minute column or the extra minutes into the hour column. Here's an improved version:

> 很明显，10点80分00秒这样的时间肯定不是你想要的结果。问题就出在了函数不值得如何应对时分秒的六十位进位，所以超过60的时候没进位就这样了。所以我们得把超出六十秒的进位到分，超过六十分的进位到小时。
>
> 下面这个是改进版本：

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second
    if sum.second >= 60:
        sum.second -= 60
        sum.minute += 1
        if sum.minute >= 60:
            sum.minute -= 60
            sum.hour += 1
    return sum
```

Although this function is correct, it is starting to get big. We will see a shorter alternative later.

> 这回函数正确工作了，但代码也开始变多了。稍后我们就能看到一个短一些的替代方案。

16.3 Modifiers 修改器 Sometimes it is useful for a function to modify the objects it gets as parameters. In that case, the changes are visible to the caller. Functions that work this way are called modifiers.

> 有时候需要对作为参数的对象进行一些修改。这时候这些修改就可以被调用者察觉。这样工作的函数就叫修改器了。

increment, which adds a given number of seconds to a Time object, can be written naturally as a modifier. Here is a rough draft:

> increment 函数，增加给定的秒数到一个 Time 对象，就可以被改写成一个修改器。
>
> 下面是个简单的版本：

```
def increment(time, seconds):
    time.second += seconds
    if time.second >= 60:
        time.second -= 60
        time.minute += 1
        if time.minute >= 60:
            time.minute -= 60
            time.hour += 1
```

The first line performs the basic operation; the remainder deals with the special cases we saw before. Is this function correct? What happens if seconds is much greater than sixty?

> 第一行代码进行了最简单的运算；后面的代码是用来应对我们之前讨论过的特例情况。
>
> 那么这个函数正确么？秒数超过六十会怎么样？

In that case, it is not enough to carry once; we have to keep doing it until time.second is less than sixty. One solution is to replace the if statements with while statements. That would make the function correct, but not very efficient.

> 很明显，秒数超过六十的时候，就需要运行不只一次了；必须一直运行，之道 time.second 的值小于六十了才行。有一种办法就是把 if 语句换成 while 语句。这样就可以解决这个问题了，但效率不太高。

As an exercise, write a correct version of increment that doesn't contain any loops.

> 做个练习，写一个正确的 increment 函数，并且要不包含任何循环。

Anything that can be done with modifiers can also be done with pure functions. In fact, some programming languages only allow pure functions. There is some evidence that programs that use pure functions are faster to develop and less error-prone than programs that use modifiers. But modifiers are convenient at times, and functional programs tend to be less efficient.

> 能用修改器实现的功能也都能用纯函数来实现。实际上有的编程语言只允许纯函数。有证据表明，与修改器相比，使用修改器能够更快地开发，而且不容易出错误。但修改器往往很方便好用，而函数式的程序一般效率要差一些。

In general, I recommend that you write pure functions whenever it is reasonable and resort to modifiers only if there is a compelling advantage. This approach might be called a functional programming style.

> 总的来说，我还是建议你写纯函数，除非用修改器有特别显著的好处。这种模式也叫做函数式编程。

As an exercise, write a "pure" version of increment that creates and returns a new Time object rather than modifying the parameter.

> 做个练习，写一个用纯函数实现的 increment，创建并返回一个新的 Time 对象，而不是修改参数。

16.4 Prototyping versus planning 原型与规划

The development plan I am demonstrating is called "prototype and patch". For each function, I wrote a prototype that performed the basic calculation and then tested it, patching errors along the way.

> 这次我演示的开发规划就是『原型与补丁模式』。对每个函数，我都先谢了一个简单的原型，只进行基本的运算，然后测试一下，接下来逐步修补错误。

This approach can be effective, especially if you don't yet have a deep understanding of the problem. But incremental corrections can generate code that is unnecessarily complicated—since it deals with many special cases—and unreliable—since it is hard to know if you have found all the errors.

> 这种模式很有效率，尤其是在你对问题的理解不是很深入的时候。不过渐进式的修改也会产生过分复杂的代码——因为要应对很多特例情况，而且也不太靠靠——因为不好确定你是否找到了所有的错误。

An alternative is designed development, in which high-level insight into the problem can make the programming much easier. In this case, the insight is that a Time object is really a three-digit number in base 60 (see Here)! The second attribute is the "ones column", the minute attribute is the "sixties column", and the hour attribute is the "thirty-six hundreds column".

> 另一种模式就是设计规划开发，这种情况下对问题的深入透彻的理解就让开发容易很多了。本节中的根本性认识就在于 TIme 对象实际上是一个三位的六十进制数字(参考 这里的解释。)！秒数也就是个位，分数也就是六十位，小时数就是三千六百位。

When we wrote add_time and increment, we were effectively doing addition in base 60, which is why we had to carry from one column to the next.

> 这样当我们写 add_time 和 increment 函数的时候，用60进制来进行计算就很有效率。

This observation suggests another approach to the whole problem—we can convert Time objects to integers and take advantage of the fact that the computer knows how to do integer arithmetic.

> 这一观察表明有另外一种方法来解决整个问题——我们可以把 Time 对象转换成整数，然后因为计算机最擅长整数运算，这样就有优势了。

Here is a function that converts Times to integers:

> 下面这个函数就把 Times 转换成了整数：

```
def time_to_int(time):
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

And here is a function that converts an integer to a Time (recall that divmod divides the first argument by the second and returns the quotient and remainder as a tuple).

> 然后下面这个函数是反过来的，把整数转换成 Time（还记得 divmod 么，使用第一个数除以第二个数，返回的是除数和余数组成的元组。）

```
def int_to_time(seconds):
    time = Time()
    minutes, time.second = divmod(seconds, 60)
    time.hour, time.minute = divmod(minutes, 60)
    return time
```

You might have to think a bit, and run some tests, to convince yourself that these functions are correct. One way to test them is to check that time_to_int(int_to_time(x)) == x for many values of x. This is an example of a consistency check.

> 你最好先考虑好了，然后多进行几次测试运行，然后要确保这些函数都是正确的。比如你就可以试着用很多个 x 的值来运算time_to_int(int_to_time(x)) == x。这也是连贯性检测的一个例子。

Once you are convinced they are correct, you can use them to rewrite add_time:

> 一旦你确定这些函数都没问题，就可以用它们来重写一下 add_time 这个函数了：

```
def add_time(t1, t2):
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

This version is shorter than the original, and easier to verify. As an exercise, rewrite increment using time_to_int and int_to_time.

> 这个版本就比最开始那个版本短多了，也更容易去检验了。接下来就做个联系吧，用 time_to_int 和 int_to_time 这两个函数来重写一下 increment。

In some ways, converting from base 60 to base 10 and back is harder than just dealing with times. Base conversion is more abstract; our intuition for dealing with time values is better.

> 在一定程度上，从六十进制到十进制的来回转换，远远比计算时间要麻烦的多。进制转换要更加抽象很多；我们处理时间计算的直觉要更好些。

But if we have the insight to treat times as base 60 numbers and make the investment of writing the conversion functions (time_to_int and int_to_time), we get a program that is shorter, easier to read and debug, and more reliable.

> 然而，如果我们有足够的远见，把时间值当做六十进制的数值来对待，然后写出一些转换函数（比如 time_to_int 和 int_to_time），就能让程序变得更短，可读性更好，调试更容易，也更加可靠。

It is also easier to add features later. For example, imagine subtracting two Times to find the duration between them. The naive approach would be to implement subtraction with borrowing. Using the conversion functions would be easier and more likely to be correct.

> 而且后续添加功能也更容易了。比如，假设要对两个时间对象进行相减来求二者之间的持续时间。简单版本的方法就是要用借位的减法。而使用转换函数的版本就更容易了，也更不容易出错。

Ironically, sometimes making a problem harder (or more general) makes it easier (because there are fewer special cases and fewer opportunities for error).

> 有意思的事，有时候以困难模式来写一个程序（比如用更加泛化的模式），反而能让开发更简单（因为这样就减少了特例情况，也减少了出错误的概率了。）

16.5 Debugging 调试

A Time object is well-formed if the values of minute and second are between 0 and 60 (including 0 but not 60) and if hour is positive. hour and minute should be integral values, but we might allow second to have a fraction part.

> 对于一个 Time 对象来说，只要分和秒的值在0-60的前闭后开区间（即可以为0但不可以为60），并且小时数为正数，就是格式正确的。小时和分钟都应该是整数，但秒是可以为小数的。

Requirements like these are called invariants because they should always be true. To put it a different way, if they are not true, something has gone wrong.

> 像这些要求也叫约束条件，因为通常都得满足这些条件才行。反过来说，如果这些条件没满足，就有可能是程序中某处存在错误了。

Writing code to check invariants can help detect errors and find their causes. For example, you might have a function like valid_time that takes a Time object and returns False if it violates an invariant:

> 写一些检测约束条件的代码，能够帮助找出这些错误，并且找到导致错误的原因。例如，你亏写一个名字未 calid_time 的函数，接收一个 Time 对象，然后如果该对象不满足约束条件就返回 False：

```
def valid_time(time):
    if time.hour < 0 or time.minute < 0 or time.second < 0:
        return False
    if time.minute >= 60 or time.second >= 60:
        return False
    return True
```

At the beginning of each function you could check the arguments to make sure they are valid:

> 然后在每个自定义函数的开头部位，你就可以检测一下参数，来确保这些参数没有错误：

```
def add_time(t1, t2):
    if not valid_time(t1) or not valid_time(t2):
        raise ValueError('invalid Time object in add_time')
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

Or you could use an assert statement, which checks a given invariant and raises an exception if it fails:

> 或者你也可以用一个 assert 语句，这个语句也是检测给定的约束条件的，如果出现错误就会抛出一个异常：

```
def add_time(t1, t2):
    assert valid_time(t1) and valid_time(t2)
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

assert statements are useful because they distinguish code that deals with normal conditions from code that checks for errors.

> assert 语句是很有用的，可以用来区分条件语句的用途，将 assert 这种用于检查错误的语句与常规的条件语句在代码上进行区分。

16.6 Glossary 术语列表

prototype and patch: A development plan that involves writing a rough draft of a program, testing, and correcting errors as they are found.

> 原型和补丁模式：一种开发模式，先写一个程序的草稿，然后测试，再改正发现的错误，这样逐步演化的开发模式。

designed development: A development plan that involves high-level insight into the problem and more planning than incremental development or prototype development.

> 设计规划开发：这种开发模式要求对所面对问题的高程度的深刻理解，相比渐进式开发和原型增补模式要更具有计划性。

pure function: A function that does not modify any of the objects it receives as arguments. Most pure functions are fruitful.

> 纯函数：不修改参数对象的函数。这种函数多数是有返回值的函数。

modifier: A function that changes one or more of the objects it receives as arguments. Most modifiers are void; that is, they return None.

> 修改器：修改参数对象的函数。大多数这样的函数都是无返回值的，也就是返回的都是None。

functional programming style: A style of program design in which the majority of functions are pure.

> 函数式编程模式：一种程序设计模式，主要特征为大多数函数都是纯函数。

invariant: A condition that should always be true during the execution of a program.

> 约束条件：在程序运行过程中，应该一直为真的条件。

assert statement: A statement that check a condition and raises an exception if it fails.

> assert 语句：一种检查错误的语句，检查一个条件，如果不满足就抛出异常。

16.7 Exercises 练习

Code examples from this chapter are available from Here; solutions to the exercises are available from Here.

> 本章的例子可以在 这里下载；练习题的答案可以在这里下载。

Exercise 1 练习1

Write a function called mul_time that takes a Time object and a number and returns a new Time object that contains the product of the original Time and the number.

> 写一个函数，名为mul_time，接收一个Time 对象和一个数值，返回一个二者相乘得到的新的Time 对象。

Then use mul_time to write a function that takes a Time object that represents the finishing time in a race, and a number that represents the distance, and returns a Time object that represents the average pace (time per mile).

> 然后用 mul_time 这个函数写一个函数，接收一个 Time 对象，代表着一个比赛的结束时间，还有一个数值，代表比赛距离，然后返回一个表示了平均步调（单位距离花费的时间）的新的 Time 对象。

Exercise 2 练习2

The datetime module provides time objects that are similar to the Time objects in this chapter, but they provide a rich set of methods and operators. Read the documentation at Here.

> datetime 模块提供了一些 time 对象，和本章的 Time 对象很相似，但前者提供了更多的方法和运算符。读一读[这里的文档] Here吧。

1. Use the datetime module to write a program that gets the current date and prints the day of the week.

> 用 datetime 模块来写一个函数，获取当前日期，然后输出今天是星期几。

1. Write a program that takes a birthday as input and prints the user's age and the number of days, hours, minutes and seconds until their next birthday.

> 写一个函数，要求输入生日，然后输出用户的年龄以及距离下一个生日的日、时、分、秒数。

1. For two people born on different days, there is a day when one is twice as old as the other. That's their Double Day. Write a program that takes two birthdays and computes their Double Day.

> 有的两个人在不同日期出生，会在某一天，一个人的年龄是另外一个人年龄的两杯。这一天就叫做他们的双倍日。写一个函数，接收两个生日，然后计算双倍日。

1. For a little more challenge, write the more general version that computes the day when one person is n times older than the other.

> 再来点有挑战性的，写一个更通用的版本，来计算一下一个人的年龄为另外一个人年龄 n 倍时候的日期。

Solution.

> 样例代码。

Chapter 17 Classes and methods 类和方法

Although we are using some of Python's object-oriented features, the programs from the last two chapters are not really object-oriented because they don't represent the relationships between programmer-defined types and the functions that operate on them. The next step is to transform those functions into methods that make the relationships explicit.

> 前两章我们已经用到了Python 的一些面向对象的特性了，但那写程序实际上并不算是真正面向对象的，因为它们并没能够表现出用户自定义类型与对这些类型进行运算的函数之间的关系。所以接下来的移步就是要把这些函数转换成方法，让这些关系更明确。

Code examples from this chapter are available from Here, and solutions to the exercises are in Here.

> 本章的样例代码可以在 这里下载，然后练习题的样例代码可以在这里下载。

17.1 Object-oriented features 面向对象的特性

Python is an object-oriented programming language, which means that it provides features that support object-oriented programming, which has these defining characteristics:

> Python 是一种面向对象的编程语言，这就意味着它提供了一些支持面向对象编程的功能，有以下这些特点：

• Programs include class and method definitions.

> 程序包含类和方法的定义。

• Most of the computation is expressed in terms of operations on objects.

> 大多数运算都以对象运算的形式来实现。

• Objects often represent things in the real world, and methods often correspond to the ways things in the real world interact.

> 对象往往代表着现实世界中的事物，方法则相对应地代表着现实世界中事物之间的相互作用。

For example, the Time class defined in Chapter 16 corresponds to the way people record the time of day, and the functions we defined correspond to the kinds of things people do with times. Similarly, the Point and Rectangle classes in Chapter 15 correspond to the mathematical concepts of a point and a rectangle.

> 例如，第16章中定义的 Time 类就代表了人们生活中计算一天时间的方法，然后当时咱们写的那些函数就对应着人们对时间的处理。同理，在第15章定义的 Point 和 Rectangle 类就对应着现实中的数学概念上的点和矩形。

So far, we have not taken advantage of the features Python provides to support object-oriented programming. These features are not strictly necessary; most of them provide alternative syntax for things we have already done. But in many cases, the alternative is more concise and more accurately conveys the structure of the program.

> 到此为止，我们还没有用到 Python 提供的用于面向对象编程的高级功能。这些高级功能并不是严格意义上必须使用的；它们大多是提供了一些我们已经实现的功能的一种备选的语法形式。不过在很多情况下，这种备选的模式更加简洁，也能更加准确地表述程序的结构。

For example, in Time1.py there is no obvious connection between the class definition and the function definitions that follow. With some examination, it is apparent that every function takes at least one Time object as an argument.

> 例如，在 Time1.py 里面，类的定义和后面的函数定义就没有啥明显的练习。测试一下就会发现，每一个后续的函数里面都至少用了一个 Time 对象作为参数。

This observation is the motivation for methods; a method is a function that is associated with a particular class. We have seen methods for strings, lists, dictionaries and tuples. In this chapter, we will define methods for programmer-defined types.

> 这样的观察结果就表明可以使用方法；方法是某一特定的类所附带的函数。之前我们看到过字符串、列表、字典以及元组的一些方法。在本章，咱们将要给用户自定义类型写一些方法。

Methods are semantically the same as functions, but there are two syntactic differences:

> 方法在语义上与函数是完全相同的，但在语法上有两点不同：

• Methods are defined inside a class definition in order to make the relationship between the class and the method explicit.

> 方法要定义在一个类定义内部，这样能保证方法和类之间的关系明确。

• The syntax for invoking a method is different from the syntax for calling a function.

> 调用一个方法的语法与调用函数的语法不一样。

In the next few sections, we will take the functions from the previous two chapters and transform them into methods. This transformation is purely mechanical; you can do it by following a sequence of steps. If you are comfortable converting from one form to another, you will be able to choose the best form for whatever you are doing.

> 在接下来的章节中，我们就要把之前两章写过的一些函数改写成方法。这种转换是纯机械的；你就遵守一系列步骤就可以实现了。如果你对二者之间的相互转化很熟悉了，你就可以根据情况自己选择是用函数还是用方法。

17.2 Printing objects 输出对象

In Chapter 16, we defined a class named Time and in Section 16.1, you wrote a function named print_time:

> 在16.1，我们定义过一个名为Time 的类，当时写过月名为 print_time 的函数：

```
class Time:
"""Represents the time of day."""
def print_time(time):
    print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

To call this function, you have to pass a Time object as an argument:

> 要调用这个函数，就必须给传递过去一个TIme 对象作为参数：

```
>>> start = Time()
>>> start = Time()
>>> start.hour = 9
>>> start.hour = 9
>>> start.minute = 45
>>> start.minute = 45
>>> start.second = 00
>>> start.second = 00
>>> print_time(start)
>>> print_time(start)
09:45:00
```

To make print_time a method, all we have to do is move the function definition inside the class definition. Notice the change in indentation.

> 要让 print_time 成为一个方法，只需要把函数定义内容放到类定义里面去。一定要注意缩进的变化哈。

```
class Time:
    def print_time(time):
        print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

Now there are two ways to call print_time. The first (and less common) way is to use function syntax:

> 现在就有两种方法来调用 print_time 这个函数了。第一种就是用函数的语法（一般大家不这么用）：

```
>>> Time.print_time(start)
>>> Time.print_time(start)
09:45:00
```

In this use of dot notation, Time is the name of the class, and print_time is the name of the method. start is passed as a parameter.

> 上面这里用到了点号，Time 是类的名字，pritn_time 是方法的名字。start 就是传过去的一个参数了。

The second (and more concise) way is to use method syntax:

> 另外一种形式就是用方法的语法（这个形式更简洁很多）：

```
>>> start.print_time()
>>> start.print_time()
09:45:00
```

In this use of dot notation, print_time is the name of the method (again), and start is the object the method is invoked on, which is called the subject. Just as the subject of a sentence is what the sentence is about, the subject of a method invocation is what the method is about.

> 在上面这里也用了点号，print_time 依然还是方法名字，然后 start 是调用方法所在的对象，也叫做主语。这里就如同句子中的主语一样，方法调用的主语就是方法的归属者。

Inside the method, the subject is assigned to the first parameter, so in this case start is assigned to time. By convention, the first parameter of a method is called self, so it would be more common to write print_time like this:

> 在方法内部，主语被用作第一个参数，所以在上面的例子中中，start 就被赋值给了 time。
>
> 按照惯例，方法的第一个参数也叫做 self，所以刚刚的 print_time 函数可以以如下这种更通用的形式来写：

```
class Time:
    def print_time(self):
        print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))
```

The reason for this convention is an implicit metaphor:

> 这种改写还有更深层次的意义：

• The syntax for a function call, print_time(start), suggests that the function is the active agent. It says something like, “Hey print_time! Here’s an object for you to print.”

> 函数调用的语法里面，print_time(start)，就表明了函数是主动的。这句语句的意思就相当于说，『嘿，print_time 函数！给你这个对象，你来打印输出一下。』

• In object-oriented programming, the objects are the active agents. A method invocation like start.print_time() says “Hey start! Please print yourself.”

> 在面向对象的编程中，对象是主动的。方法的调用，比如 start.rint_time()，就像是说，『嘿，start，你打印输出一下你自己』

This change in perspective might be more polite, but it is not obvious that it is useful. In the examples we have seen so far, it may not be. But sometimes shifting responsibility from the functions onto the objects makes it possible to write more versatile functions (or methods), and makes it easier to maintain and reuse code.

> 看上去这样改了之后客气了不少，实际上不止如此，还有更多用处，只是不太明显。目前我们看到过的例子里面，这样改写还没有造成什么区别。但是有的时候，从函数转为对象，能够让函数（或者方法）更加通用，也让程序更容易维护，还便于代码的重用。

As an exercise, rewrite time_to_int (from Section 16.4) as a method. You might be tempted to rewrite int_to_time as a method, too, but that doesn’t really make sense because there would be no object to invoke it on.

> 做个练习吧，重写一下 time_to_int（参见16.4），把这个函数写成一个方法。你也可以试着把 int_to_time 也携程方法，不过这可能不太行得通，因为把这个函数改成方法的话，没有对象来调用方法。

17.3 Another example 另外一个例子

Here’s a version of increment (from Section 16.3) rewritten as a method:

> 下面是 increment 函数（参见16.4）被改写成的方法：

```
# inside class Time:
def increment(self, seconds):
    seconds += self.time_to_int()
    return int_to_time(seconds)
```

This version assumes that time_to_int is written as a method. Also, note that it is a pure function, not a modifier. Here’s how you would invoke increment:

> 这一版本的前提是 time_to_int 已经被改写成方法了。另外也要注意到，这是一个纯函数，而不是修改器。
>
> 下面是调用 increment 的示范：

```
>>> start.print_time()
>>> start.print_time()
09:45:00
>>> end = start.increment(1337)
>>> end = start.increment(1337)
>>> end.print_time()
>>> end.print_time()
10:07:17
```

The subject, start, gets assigned to the first parameter, self. The argument,1337, gets assigned to the second parameter, seconds. This mechanism can be confusing, especially if you make an error. For example, if you invoke increment with two arguments, you get:

> 主语，start，用自己（self）赋值给第一个参数。然后参数，1337，赋值给了第二个参数，秒值seconds。
>
> 这种表述挺混乱，如果弄错了就更麻烦了。比如，如果你用两个参数调用了 increment 函数，你会得到如下的错误：

```
>>> end = start.increment(1337, 460)
>>> end = start.increment(1337, 460)
TypeError: increment() takes 2 positional arguments but 3 were given
```

The error message is initially confusing, because there are only two arguments in parentheses. But the subject is also considered an argument, so all together that's three.

> 这个错误信息刚开始看上去还挺不好理解，因为括号里面确实是只有两个参数。不过实际上主语也会把自己当做一个参数，所以总共实际上是有三个参数了。

By the way, a positional argument is an argument that doesn't have a parameter name; that is, it is not a keyword argument. In this function call:

> 另外，有一种参数叫位置参数，就是没有参数的名字；这种参数就和关键字参数不同了。下面这个函数调用中：

```
sketch(parrot, cage, dead=True)
```

parrot and cage are positional, and dead is a keyword argument.

> parrot 和 cage 都是位置参数，dead 是关键字参数。

17.4 A more complicated example 更复杂点的例子

Rewriting is_after (from Section 16.1) is slightly more complicated because it takes two Time objects as parameters. In this case it is conventional to name the first parameter self and the second parameter other:

> 重写 is_after（参见16.1），这就比较有难度了，因为这个函数接收两个 Time 对象作为参数。在这个情况下，一般就把第一个参数命名为 self，第二个命名为 other：

```
# inside class Time:
def is_after(self, other):
    return self.time_to_int() > other.time_to_int()
```

To use this method, you have to invoke it on one object and pass the other as an argument:

> 要使用这个方法，就必须在一个对象后面调用，然后用另外一个对象作为参数：

```
>>> end.is_after(start)
>>> end.is_after(start)
True
```

One nice thing about this syntax is that it almost reads like English: “end is after start?”

> 这里就体现出一种语法上的好处了，因为读起来基本就根英语是一样的嗯：『end is after start?』

17.5 The init method init方法

The init method (short for “initialization”) is a special method that gets invoked when an object is instantiated. Its full name is __init__ (two underscore characters, followed by init, and then two more underscores). An init method for the Time class might look like this:

> init 方法（就是对『initialization』的缩写，初始化的意思，这个方法相当于C++中的构造函数）是一种特殊的方法，在对象被实例化的时候被调用。这个方法的全名是 __init__（两个下划线，然后是 init，然后还是两个下划线）。在 Time 类当中，init 方法示例如下：

```
# inside class Time:
def __init__(self, hour=0, minute=0, second=0):
    self.hour = hour
    self.minute = minute
    self.second = second
```

It is common for the parameters of __init__ to have the same names as the attributes. The statement

> 一般情况下，__init__ 方法里面的参数与属性变量的名字是相同的。下面这个语句

```
        self.hour = hour
```

stores the value of the parameter hour as an attribute of self. 就存储了参数 hour 的值，赋给了属性变量hour本身。 The parameters are optional, so if you call Time with no arguments, you get the default values.

> 这些参数都是可选的，所以如果你调用 Time 但不给任何参数，得到的就是默认值。

```
>>> time = Time()
>>> time = Time()
>>> time.print_time()
>>> time.print_time()
00:00:00
```

If you provide one argument, it overrides hour:

> 如果你提供一个参数，就先覆盖 hour 的值：

```
>>> time = Time (9)
>>> time = Time (9)
>>> time.print_time()
>>> time.print_time()
09:00:00
```

If you provide two arguments, they override hour and minute.

> 提供两个参数，就先后覆盖了 hour 和 minute 的值。

```
>>> time = Time(9, 45)
>>> time = Time(9, 45)
>>> time.print_time()
>>> time.print_time()
09:45:00
```

And if you provide three arguments, they override all three default values.

> 如果你给出三个参数，就依次覆盖掉所有三个默认值了。

As an exercise, write an init method for the Point class that takes x and y as optional parameters and assigns them to the corresponding attributes.

> 做一个练习，写一个 Point 类的 init 方法，接收 x 和 y 作为可选参数，然后赋值给对应的属性。

17.6 The __str__ method str 方法

__str__ is a special method, like __init__, that is supposed to return a string representation of an object.

> __str__ 是一种特殊的方法，就跟__init__差不多，str 方法是接收一个对象，返回一个代表该对象的字符串。

For example, here is a str method for Time objects:

> 例如，下面就是Time 对象的一个 str 方法：

```
# inside class Time:
def __str__(self):
    return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

When you print an object, Python invokes the str method:

> 这样当你用 print 打印输出一个对象的时候，Python 就会调用这个 str 方法：

```
>>> time = Time(9, 45)
>>> time = Time(9, 45)
>>> print(time) 09:45:00
>>> print(time) 09:45:00
```

When I write a new class, I almost always start by writing __init__, which makes it easier to instantiate objects, and __str__, which is useful for debugging.

> 写一个新的类的时候，总要先写出来 __init__ 方法，这样有利于简化对象的初始化，还要写个 __str__ 方法，这个方法在调试的时候很有用。

As an exercise, write a str method for the Point class. Create a Point object and print it.

> 做个练习，写一下 Point 这个类的 str 方法。创建一个 Point 对象，然后用 print 输出一下。

17.7 Operator overloading 运算符重载

By defining other special methods, you can specify the behavior of operators on programmer-defined types. For example, if you define a method named__add__ for the Time class, you can use the + operator on Time objects. Here is what the definition might look like:

> 通过定义一些特定的方法，咱们就能针对自定义类型，让运算符有特定的作用。比如，如果你在 Time 类中定义了一个名字为__add__的方法，你就可以对 Time 对象使用『+』加号运算符。

```
# inside class Time:
def __add__(self, other):
    seconds = self.time_to_int() + other.time_to_int()
    return int_to_time(seconds)
```

And here is how you could use it:

> 使用方法如下所示：

```
>>> start = Time(9, 45)
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> duration = Time(1, 35)
>>> print(start + duration)
>>> print(start + duration)
11:20:00
```

When you apply the + operator to Time objects, Python invokes __add__. When you print the result, Python invokes __str__. So there is a lot happening behind the scenes!

> 当你针对 Time 对象使用加号运算符的时候，Python 就会调用你刚刚自定义的 __add__ 方法。当你用 print 输出结果的时候，Python 调用的是你自定义的 __str__ 方法。所以实际上面这个简单的例子背后可不简单。

Changing the behavior of an operator so that it works with programmer-defined types is called operator overloading. For every operator in Python there is a corresponding special method, like __add__. For more details, see Here.

> 针对用户自定义类型，让运算符有相应的行为，这就叫做运算符重载。Python 当中每一个运算符都有一个对应的方法，比如__add__。更多内容可以看一下 这里的文档。

As an exercise, write an add method for the Point class.

> 做个练习，给 Point 类写一个加法的方法。

17.8 Type-based dispatch 根据对象类型进行运算

In the previous section we added two Time objects, but you also might want to add an integer to a Time object. The following is a version of __add__ that checks the type of other and invokes either add_time or increment:

> 在前面的章节中，我们把两个 Time 对象进行了相加，但也许有时候需要把一个整数加到 Time 对象上面。下面这一个版本的__add__方法就能够实现检查类型，然后调用 add_time 方法或者是 increment 方法：

```
# inside class Time:
def __add__(self, other):
    if isinstance(other, Time):
        return self.add_time(other)
    else:
        return self.increment(other)
def add_time(self, other):
    seconds = self.time_to_int() + other.time_to_int()
    return int_to_time(seconds)
def increment(self, seconds):
    seconds += self.time_to_int()
    return int_to_time(seconds)
```

The built-in function isinstance takes a value and a class object, and returns True if the value is an instance of the class.

> 内置函数isinstance 接收一个值和一个类的对象，如果该值是这个类的一个实例，就会返回真。

If other is a Time object, __add__ invokes add_time. Otherwise it assumes that the parameter is a number and invokes increment. This operation is called a type-based dispatch because it dispatches the computation to different methods based on the type of the arguments.

> 如果拿来相加的是一个 Time 对象，__add__就会调用 add_time 方法。其他情况下，程序会把参数当做一个数字，然后就调用 increment 方法。这种运算就是根据对象进行的，因为在针对不同类型参数的时候，运算符会进行不同的计算。

Here are examples that use the + operator with different types:

> 下面的例子中，就展示了用不同类型变量来相加的效果：

```
>>> start = Time(9, 45)
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> duration = Time(1, 35)
>>> print(start + duration)
>>> print(start + duration)
11:20:00
>>> print(start + 1337)
>>> print(start + 1337)
10:07:17
```

Unfortunately, this implementation of addition is not commutative. If the integer is the first operand, you get

> 然而不幸的是，这个加法运算不满足交换率。如果整数放到首位，就会得到如下所示的错误了：

```
>>> print(1337 + start)
>>> print(1337 + start)
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

The problem is, instead of asking the Time object to add an integer, Python is asking an integer to add a Time object, and it doesn’t know how. But there is a clever solution for this problem: the special method __radd__, which stands for “right-side add”. This method is invoked when a Time object appears on the right side of the + operator. Here’s the definition:

> 这里的问题就在于，Python 并没有让一个 Time 对象来加一个整数，而是去调用了整形的加法去把一个 Time 对象加到整数上面去，这就用系统原本的加法，而这个加法不能处理 Time 对象。有一个很聪明的方法来解决这个问题：用一个特殊的方法__radd__，这个方法的意思就是『右加』。在一个 Time 对象出现在加号运算符右侧的时候，该方法就会被调用了。下面就是这个方法的定义：

```
# inside class Time:
def __radd__(self, other):
    return self.__add__(other)
```

And here's how it's used:

> 使用如下所示：

```
>>> print(1337 + start)
>>> print(1337 + start)
10:07:17
```

As an exercise, write an add method for Points that works with either a Point object or a tuple:

> 做个练习，为 Point 类来写一个加法的方法，要求能处理Point 对象或者一个元组：

• If the second operand is a Point, the method should return a new Point whose x coordinate is the sum of the x coordinates of the operands, and likewise for the y coordinates.

> 如果第二个运算数是一个Point，该方法就应该返回一个新的Point，新点的横纵坐标分别为两个点坐标相加。

• If the second operand is a tuple, the method should add the first element of the tuple to the x coordinate and the second element to the y coordinate, and return a new Point with the result.

> 如果第二个运算数是一个元组，该方法就要把元组中第一个元素加到横坐标上，把第二个元素加到纵坐标上面，然后用计算出来的坐标返回一个新的点。

17.9 Polymorphism 多态

Type-based dispatch is useful when it is necessary, but (fortunately) it is not always necessary. Often you can avoid it by writing functions that work correctly for arguments with different types.

> 在必要的时候，根据类型运算还是很有用的，不过（还好）并不总需要这么做。一般你都可以把函数写成能处理不同类型参数的，这样就不用这么麻烦了。

Many of the functions we wrote for strings also work for other sequence types. For example, in Section 11.2 we used histogram to count the number of times each letter appears in a word.

> 我们之前为字符串写的很多函数，也都可以用到其他序列类型上面。比如在11.2我们用 histogram 来统计一个单词中每个字母出现的次数。

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] = d[c]+1
    return d
```

This function also works for lists, tuples, and even dictionaries, as long as the elements of s are hashable, so they can be used as keys in d.

> 这个函数也可以用于列表、元组，甚至字典，只要s 的元素是散列的，就能用做 d 当中的键。

```
>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> histogram(t)
>>> histogram(t)
{'bacon': 1, 'egg': 1, 'spam': 4}
```

Functions that work with several types are called polymorphic. Polymorphism can facilitate code reuse. For example, the built-in function sum, which adds the elements of a sequence, works as long as the elements of the sequence support addition.

> 针对不同类型都可以运行的函数，就是多态的了。多态能够有利于代码复用。比如内置的函数 sum，是用来把一个序列中所有的元素加起来，就可以适用于所有能够相加的序列元素。

Since Time objects provide an add method, they work with sum:

> Time 对象有了自己的加法方法，就可以与 sum 函数来配合使用了：

```
>>> t1 = Time(7, 43)
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 41)
>>> t2 = Time(7, 41)
>>> t3 = Time(7, 37)
>>> t3 = Time(7, 37)
>>> total = sum([t1, t2, t3])
>>> total = sum([t1, t2, t3])
>>> print(total)
>>> print(total)
23:01:00
```

In general, if all of the operations inside a function work with a given type, the function works with that type. The best kind of polymorphism is the unintentional kind, where you discover that a function you already wrote can be applied to a type you never planned for.

> 总的来说，如果一个函数内的所有运算都可以处理某一类型，这个函数就适用于这一类型了。
>
> 最好就是无心插柳柳成荫的这种多态，这种情况下你会发现某个之前写过的函数可以用来处理一个之前没有计划到的类型。

17.10 Debugging 调试

It is legal to add attributes to objects at any point in the execution of a program, but if you have objects with the same type that don't have the same attributes, it is easy to make mistakes. It is considered a good idea to initialize all of an object's attributes in the init method.

> 在程序运行的任意时刻都可以给对象增加属性，不过如果你有多个同类对象却又不具有相同的属性，就容易出错了。所以最好在对象实例化的时候就全部用 init 方法初始化对象的全部属性。

If you are not sure whether an object has a particular attribute, you can use the built-in function hasattr (see Section 15.7).

> 如果你不确定一个对象是否有某个特定的属性，你可以用内置的 hasattr 函数来尝试一下（参考15.7）。

Another way to access attributes is the built-in function vars, which takes an object and returns a dictionary that maps from attribute names (as strings) to their values:

> 另外一种读取属性的方法是用内置函数 vars，这个函数会接收一个对象，然后返回一个字典，字典中的键值对就是属性名的字符串与对应的值。

```
>>> p = Point(3, 4)
>>> p = Point(3, 4)
>>> vars(p)
>>> vars(p)
{'y': 4, 'x': 3}
```

For purposes of debugging, you might find it useful to keep this function handy:

> 出于调试目的，你估计也会发现下面这个函数随时用一下会带来很多便利：

```
def print_attributes(obj):
    for attr in vars(obj):
        print(attr, getattr(obj, attr))
```

print_attributes traverses the dictionary and prints each attribute name and its corresponding value. The built-in function getattr takes an object and an attribute name (as a string) and returns the attribute's value.

> print_attributes 会遍历整个字典，然后输出每一个属性的名字与对应的值。
>
> 内置函数 getattr 会接收一个对象和一个属性名字（以字符串形式），然后返回该属性的值。

17.11 Interface and implementation 接口和实现

One of the goals of object-oriented design is to make software more maintainable, which means that you can keep the program working when other parts of the system change, and modify the program to meet new requirements. A design principle that helps achieve that goal is to keep interfaces separate from implementations. For objects, that means that the methods a class provides should not depend on how the attributes are represented.

> 面向对象编程设计的目的之一就是让软件更容易维护，这就意味着当系统中其他部分发生改变的时候依然能让程序运行，然后可以修改程序去符合新的需求。
>
> 实现这一目标的程序设计原则就是要让接口和实现分开。对于对象来说，这就意味着一个类包含的方法要不能被属性表达方式的变化所影响。

For example, in this chapter we developed a class that represents a time of day. Methods provided by this class include time_to_int, is_after, and add_time.

> 比如，在本章我们建立了一个表示一天中时间的类。该类提供的方法包括 time_to_int, is_after, 和 add_time。

We could implement those methods in several ways. The details of the implementation depend on how we represent time. In this chapter, the attributes of a Time object are hour, minute, and second.

> 我们可以用几种不同方式来实现这些方法。这些实现的细节依赖于我们如何去表示时间。在本章，一个 Time 对象的属性为时分秒三个变量。

As an alternative, we could replace these attributes with a single integer representing the number of seconds since midnight. This implementation would make some methods, like is_after, easier to write, but it makes other methods harder.

> 还有一种替代方案，我们就可以把这些属性替换为一个单个的整形变量，表示从午夜零点到当前时间的秒的数目。这种实现方法可以让一些方法更简单，比如 is_after，但也让其他方法更难写了。

After you deploy a new class, you might discover a better implementation. If other parts of the program are using your class, it might be time-consuming and error-prone to change the interface.

> 当你创建一个新的类之后，你可能会发现有更好的实现方式。如果一个程序的其他部位在用你的类，这时候再来改造接口可能就要消耗很多时间，也容易遇到很多错误了。

But if you designed the interface carefully, you can change the implementation without changing the interface, which means that other parts of the program don't have to change.

> 但如果你仔细地设计好接口，你在改变实现的时候就不用去折腾了，这就意味着你程序的其他部位都不需要改动了。

17.12 Glossary 术语列表

object-oriented language: A language that provides features, such as programmer-defined types and methods, that facilitate object-oriented programming.

> 面向对象的编程语言：提供面向对象功能的语言，比如用户自定义类型和方法，有利于实现面向对象编程。

object-oriented programming: A style of programming in which data and the operations that manipulate it are organized into classes and methods.

> 面向对象编程：一种编程模式，数据和运算都被封装进类和方法之中。

method: A function that is defined inside a class definition and is invoked on instances of that class.

> 方法：类内所包含的函数就叫方法，可以在类的接口中被调用。

subject: The object a method is invoked on.

> 主语：调用方法的对象。

positional argument: An argument that does not include a parameter name, so it is not a keyword argument.

> 位置参数：一种参数，没有参数名字，不是关键字参数。

operator overloading: Changing the behavior of an operator like + so it works with a programmer-defined type.

> 运算符重载：像+加号这样的运算符，在处理用户自定义类型的时候改变为相应的运算。

type-based dispatch: A programming pattern that checks the type of an operand and invokes different functions for different types.

> 按类型处理：一种编程模式，检查运算数的类型，然后根据类型调用不同的函数来进行运算。

polymorphic: Pertaining to a function that can work with more than one type.

> 多态：一个函数能处理多种类型的特征，就叫做多态。

information hiding: The principle that the interface provided by an object should not depend on its implementation, in particular the representation of its attributes.

> 信息隐藏：一种开发原则，一个对象提供的接口应该独立于其实现，尤其是不受对象属性设置变化的影响。

17.13 Exercises 练习

Exercise 1 练习1

Download the code from this chapter from Here. Change the attributes of Time to be a single integer representing seconds since midnight. Then modify the methods (and the function int_to_time) to work with the new implementation. You should not have to modify the test code in main. When you are done, the output should be the same as before. Solution

> 从这里下载本章的代码。把 Time 中的属性改变成一个单独的整型变量，用来表示自从午夜至今的秒数。然后修改一下各个方法（以及 int_to_time 函数），让所有功能都能在新的实现下正常工作。尽量就让自己不用去更改main 当中的测试代码。你改完之后，输出应该与之前相同。样例代码

Exercise 2 练习2

This exercise is a cautionary tale about one of the most common, and difficult to find, errors in Python. Write a definition for a class named Kangaroo with the following methods:

> 这个练习是一个广为流传的寓言故事，其中包含了一个使用 Python的时候最常见但也是最难发现的错误。写一个名为袋鼠的类的定义，要求有如下的方法：

1. An __init__ method that initializes an attribute named pouch_contents to an empty list.

> 一个__init__方法，用来初始化一个名为 puntch_contents（就是袋鼠的袋子中内容的意思） 的属性，把该属性初始化为一个空列表。

1. A method named put_in_pouch that takes an object of any type and adds it to pouch_contents.

> 一个名为put_in_pouch 的方法，接收任意类型的一个对象，把这个对象放进 pouch_contents 中。

1. A __str__ method that returns a string representation of the Kangaroo object and the contents of the pouch.

> 一个__str__方法，返回袋鼠对象的字符串表示，以及袋鼠袋子中的内容。

Test your code by creating two Kangaroo objects, assigning them to variables named kanga and roo, and then adding roo to the contents of kanga's pouch.

> 通过建立两个袋鼠对象来测试一下你的代码，把它们俩分别命名为 kanga 和 roo，然后把roo 添加到 kanga 的袋子中。

Download This. It contains a solution to the previous problem with one big, nasty bug. Find and fix the bug.

> 下载这个代码。里面包含了上面这个练习的一个样例代码，但这个代码有很大很悲催的 bug。找出这个 bug 然后改过来吧。

If you get stuck, you can download This, which explains the problem and demonstrates a solution.

> 如果你搞不定了，可以下载这个代码，这个代码中解释了整个问题，并且提供了一个可行的解决方案。

Chapter 18 Inheritance 继承

The language feature most often associated with object-oriented programming is inheritance. Inheritance is the ability to define a new class that is a modified version of an existing class. In this chapter I demonstrate inheritance using classes that represent playing cards, decks of cards, and poker hands.

> 面向对象编程最常被人提到的语言功能就是继承了。继承就是基于一个已有的类进行修改来定义一个新的类。在本章我会用一些例子来演示继承，这些例子会用到一些类来表示扑克牌，成副的纸牌和扑克牌型。

If you don't play poker, you can read about it at Here, but you don't have to; I'll tell you what you need to know for the exercises.

> 如果你没玩过扑克，你可以读一下这里的介绍，不过也没必要；因为我等会会把练习中涉及到的相关内容给你解释明白的。

Code examples from this chapter are available from Here.

> 本章的代码样例可以在这里下载。

18.1 Card objects 纸牌对象

There are fifty-two cards in a deck, each of which belongs to one of four suits and one of thirteen ranks. The suits are Spades, Hearts, Diamonds, and Clubs (in descending order in bridge). The ranks are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, and King. Depending on the game that you are playing, an Ace may be higher than King or lower than 2.

> 牌桌上面一共有52张扑克牌，每一张都属于四种花色之一，并且是十三张牌之一。花色为黑桃，红心，方块，梅花（在桥牌中按照降序排列）。排列顺序为 A，2，3，4，5，6，7，8，9，10，J，Q，K。根据具体玩的游戏的不同，A 可以比 K 大，也可以比2还小。

If we want to define a new object to represent a playing card, it is obvious what the attributes should be: rank and suit. It is not as obvious what type the attributes should be. One possibility is to use strings containing words like 'Spade' for suits and 'Queen' for ranks. One problem with this implementation is that it would not be easy to compare cards to see which had a higher rank or suit.

> 如果咱们要定义一个新的对象来表示一张牌，很明显就需要两个属性了：点数以及花色。但这两个属性应该是什么类型呢，就不那么明显了。一种思路是用字符串，就比如用『黑桃』来表示花色，『Q』来表示点数。不过这个实现方法不怎么方便，不好去比较纸牌的点数大小以及花色。

An alternative is to use integers to encode the ranks and suits. In this context, “encode” means that we are going to define a mapping between numbers and suits, or between numbers and ranks. This kind of encoding is not meant to be a secret (that would be “encryption”).

> 另外一种思路是用整数来编码，以表示点数和花色。在这里，『编码』的意思就是我们要建立一个从数值到花色或者从数值到点数的映射。这种编码并不是为了安全的考虑（那种情况下用的词是『encryption（也是编码的意思，专用于安全领域）』）。

For example, this table shows the suits and the corresponding integer codes:

> 例如，下面这个表格就表示了花色与整数编码之间的映射关系：

```
Spades    ↦    3
Hearts    ↦    2
Diamonds    ↦    1
Clubs    ↦    0
```

This code makes it easy to compare cards; because higher suits map to higher numbers, we can compare suits by comparing their codes.

> 这样的编码就比较易于比较牌的大小；因为高花色对应着大数值，我们对比一下编码大小就能比较花色顺序。

The mapping for ranks is fairly obvious; each of the numerical ranks maps to the corresponding integer, and for face cards:

> 牌面大小的映射就很明显了；每一张牌都对应着相应大小的整数，对于有人像的几张映射如下所示：

```
Jack    ↦    11
Queen    ↦    12
King    ↦    13
```

I am using the ↦ symbol to make it clear that these mappings are not part of the Python program. They are part of the program design, but they don’t appear explicitly in the code.

> 我这里用箭头符号 ↦ 来表示映射关系，但这个符号并不是 Python 所支持的。这些符号是程序设计的一部分，但最终并不以这种形式出现在代码里。

The class definition for Card looks like this:

> 这样实现的纸牌类的定义如下所示：

```
class Card:
"""Represents a standard playing card."""
    def __init__(self, suit=0, rank=2):
        self.suit = suit
        self.rank = rank
```

As usual, the init method takes an optional parameter for each attribute. The default card is the 2 of Clubs. To create a Card, you call Card with the suit and rank of the card you want.

> 一如既往，init 方法可以为每一个属性接收一个可选参数来初始化。默认的牌面为梅花2.
>
> 要建立一张纸牌，可以用你想要的花色和牌值调用 Card。

```
queen_of_diamonds = Card(1, 12)
```

18.2 Class attributes 类的属性

In order to print Card objects in a way that people can easily read, we need a mapping from the integer codes to the corresponding ranks and suits. A natural way to do that is with lists of strings. We assign these lists to class attributes:

> 想要以易于被人理解的方式来用 print 打印输出纸牌对象，我们就得建立一个从整形编码到对应的牌值和花色的映射。最自然的方法莫过于用字符串列表来实现。咱们可以先把这些列表赋值到类的属性中去：

```
# inside class Card:
suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7','8', '9', '10', 'Jack', 'Queen',
def __str__(self):
    return '%s of %s' % (Card.rank_names[self.rank],Card.suit_names[self.suit])
```

Variables like suit_names and rank_names, which are defined inside a class but outside of any method, are called class attributes because they are associated with the class object Card.

> suit_names 和 rank_names 这样的变量，都是在类内定义，但不在任何方法之内，这就叫做类的属性，因为它们属于类 Card。

This term distinguishes them from variables like suit and rank, which are called instance attributes because they are associated with a particular instance.

> 这种形式就把类的属性与变量 suit 和 rank 区分开来，后面这两个变量叫做实例属性，因为这两个属性取决于具体的实例。

Both kinds of attribute are accessed using dot notation. For example, in__str__, self is a Card object, and self.rank is its rank. Similarly, Card is a class object, and Card.rank_names is a list of strings associated with the class.

> 这些属性都可以用点号来读取。比如，在__str__方法中，self 是一个 Card 对象，而 self.rank 就是该对象的 rank 变量。同理，Card 是一个class 对象，而 Card.rank_names 就是属于该类的一个字符串列表。

Every card has its own suit and rank, but there is only one copy of suit_names and rank_names.

> 没一张牌都有自己的花色和牌值，但都只有唯一的一套 suit_names 和 rank_names。

Putting it all together, the expression Card.rank_names[self.rank] means “use the attribute rank from the object self as an index into the list rank_names from the class Card, and select the appropriate string.”

> 放到一起，这个表达式Card.rank_names[self.rank]的意思就是『用对象 self 的 rank 属性作为一个索引，从类 Card 中的rank_names 列表中选择该索引位置的字符串。』

The first element of rank_names is None because there is no card with rank zero. By including None as a place-keeper, we get a mapping with the nice property that the index 2 maps to the string '2', and so on. To avoid this tweak, we could have used a dictionary instead of a list.

> rank_names 的一个元素是None空，因为没有牌值为0的纸牌。包含 None 在内作为一个替位符，整个映射就很简明，索引2的位置对应着就是字符串「2」，其他牌值依此类推。要是觉得这样太别扭，咱们还可以用字典来替代列表。

With the methods we have so far, we can create and print cards:

> 目前已经有了这些方法了，咱们就可以创建和打印输出纸牌了：

```
>>> card1 = Card(2, 11)
>>> card1 = Card(2, 11)
>>> print(card1)
>>> print(card1)
Jack of Hearts
```

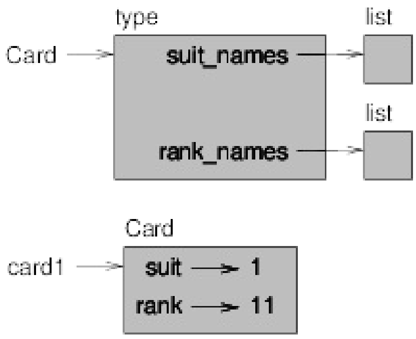


Figure 18.1: Object diagram.

Figure 18.1 is a diagram of the Card class object and one Card instance. Card is a class object; its type is type. card1 is an instance of Card, so its type is Card. To save space, I didn't draw the contents of suit_names and rank_names.

> 图18.1是一个 Card 类对象以及一个 Card 实例的图解。Card 是一个类对象（就是类的一个实例）；它的类型是type。card1是 Card 的一个实例，所以它的类型是 Card。为了节省空间，我没有画出 suit_names 和 rank_names 的内容。

18.3 Comparing cards 对比牌值

For built-in types, there are relational operators (<, >, ==, etc.) that compare values and determine when one is greater than, less than, or equal to another. For programmer-defined types, we can override the behavior of the built-in operators by providing a method named __lt__, which stands for "less than".

> 对于内置类型，直接就可以用关系运算符（<, >, ==,等等）比较两个值来判断二者的大小以及是否相等。对与用户自定义类型，咱们就要覆盖掉内置运算符的行为，这就需要提供一个名为__lt__的方法，这个lt 就是『less than』的缩写，意思是『小于』。

__lt__ takes two parameters, self and other, and True if self is strictly less than other. The correct ordering for cards is not obvious. For example, which is better, the 3 of Clubs or the 2 of Diamonds? One has a higher rank, but the other has a higher suit. In order to compare cards, you have to decide whether rank or suit is more important.

> __lt__接收两个参数，一个是self，一个是另外一个对象，如果 self 严格小于另外一个对象，就返回真。
>
> 纸牌的牌值大小排列并不是很简单。比如，梅花3和方块2哪个更大呢？一个的牌值更高，但另一个的花色更高。所以要进行比较的话，你就得确定牌值和花色哪个更重要。

The answer might depend on what game you are playing, but to keep things simple, we'll make the arbitrary choice that suit is more important, so all of the Spades outrank all of the Diamonds, and so on.

> 实际上这种关系还得取决于你玩的纸牌游戏中的规则，不过为了简单起见，咱们就做一个武断的选择，就让花色更重要，所以所有的黑桃都大于方块，依此类推了。

With that decided, we can write __lt__:

> 确定好规则了，就可以写这个__lt__方法了：

```
# inside class Card:
def __lt__(self, other):
# check the suits
    if self.suit < other.suit:
        return True
    if self.suit > other.suit:
        return False
# suits are the same... check ranks
    return self.rank < other.rank
```

You can write this more concisely using tuple comparison:

> 用元组对比就可以把代码写得更简洁了：

```
# inside class Card:
def __lt__(self, other):
    t1 = self.suit, self.rank
    t2 = other.suit, other.rank
    return t1 < t2
```

As an exercise, write an __lt__ method for Time objects. You can use tuple comparison, but you also might consider comparing integers.

> 做个练习，为 Time 对象写一个__lt__方法。可以用元组对比，不过也可以对比整数。

18.4 Decks 成副的纸牌

Now that we have Cards, the next step is to define Decks. Since a deck is made up of cards, it is natural for each Deck to contain a list of cards as an attribute.

> 现在咱们已经有了纸牌的类了，接下来的一不就是定义成副纸牌了。因为一副纸牌上是有各种牌，所以很自然就应该包含一个纸牌列表作为一个属性了。

The following is a class definition for Deck. The init method creates the attribute cards and generates the standard set of fifty-two cards:

> 下面就是一个一副纸牌类的定义。init 方法建立了一个属性 cards，然后生成了标准的五十二张牌来初始化。

```
class Deck:
    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1, 14):
                card = Card(suit, rank)
                self.cards.append(card)
```

The easiest way to populate the deck is with a nested loop. The outer loop enumerates the suits from 0 to 3. The inner loop enumerates the ranks from 1 to 13. Each iteration creates a new Card with the current suit and rank, and appends it to self.cards.

> 实现一副牌的最简单方法就是用网状循环了。外层循环枚举花色从0到3一共四种。内层的循环枚举从1到13的所有牌值。每一次循环都以当前的花色和牌值创建一个新的Card对象，添加到 self.cards 列表中。

18.5 Printing the deck 输出整副纸牌

Here is a __str__ method for Deck:

> 下面是 Deck 类的__str__方法：

```
#inside class Deck:
def __str__(self):
    res = []
    for card in self.cards:
        res.append(str(card))
    return '\n'.join(res)
```

This method demonstrates an efficient way to accumulate a large string: building a list of strings and then using the string method join. The built-in function str invokes the __str__ method on each card and returns the string representation.

> 上面的方法展示了累积大字符串的一种有效方法：建立一个字符串列表，然后用字符串方法 join 实现。内置函数 str 调用每一张牌的__str__方法，然后返回该张纸牌的字符串表示。

Since we invoke join on a newline character, the cards are separated by newlines. Here's what the result looks like:

> 由于我们调用 join的位置在换行符后面，这样这些纸牌就被换行符分开了。程序运行结果如下所示：

```
>>> deck = Deck()
>>> deck = Deck()
>>> print(deck)
>>> print(deck)
Ace of Clubs
2 of Clubs
3 of Clubs
...
10 of Spades
Jack of Spades
Queen of Spades
King of Spades
```

Even though the result appears on 52 lines, it is one long string that contains newlines.

> 虽然结果看上去是52行，但实际上只是一个包含了很多换行符的一个长字符串。

18.6 Add, remove, shuffle and sort 添加，删除，洗牌和排序

To deal cards, we would like a method that removes a card from the deck and returns it. The list method pop provides a convenient way to do that:

> 要处理纸牌，我们还需要一个方法来从牌堆中拿出和放入纸牌。列表的 pop 方法很适合来完成这件任务：

```
#inside class Deck:
def pop_card(self):
    return self.cards.pop()
```

Since pop removes the last card in the list, we are dealing from the bottom of the deck. To add a card, we can use the list method append:

> pop 方法从列表中拿走最后一张牌，这样就是从一副牌的末尾来处理。
>
> 要添加一张牌，可以用列表的 append 方法：

```
#inside class Deck:
def add_card(self, card):
    self.cards.append(card)
```

A method like this that uses another method without doing much work is sometimes called a veneer. The metaphor comes from woodworking, where a veneer is a thin layer of good quality wood glued to the surface of a cheaper piece of wood to improve the appearance.

> 上面这种方法都是调用了其他的方法，而没有做什么别的事情，所以也被叫做镶板。这个比喻来自于木匠行业，镶板就是一薄层的高端木料用胶水贴到廉价木料上面，来提高视觉效果。

In this case add_card is a “thin” method that expresses a list operation in terms appropriate for decks. It improves the appearance, or interface, of the implementation.

> 在刚刚的例子中，add_card 就相当于那个『高端』的方法，表示的是适用于处理纸牌的列表操作。这样就提高了程序实现的可读性，或者说改善了接口。

As another example, we can write a Deck method named shuffle using the function shuffle from the random module:

> 再举一个例子，咱们再来给 Deck写一个洗牌的方法，用 random（随机的意思）模块的 shuffle 方法：

```
# inside class Deck:
def shuffle(self):
    random.shuffle(self.cards)
```

Don’t forget to import random. As an exercise, write a Deck method named sort that uses the list method sort to sort the cards in a Deck. sort uses the __lt__ method we defined to determine the order.

> 一定别忘了导入 random 模块。
>
> 做个练习吧，写一个名为 sort 的方法给 Deck，使用列表的sort 方法来给 Deck 中的牌进行排序。sort 方法要用到我们之前写过的 __lt__ 方法来确定顺序。

18.7 Inheritance 继承

Inheritance is the ability to define a new class that is a modified version of an existing class. As an example, let’s say we want a class to represent a “hand”, that is, the cards held by one player. A hand is similar to a deck: both are made up of a collection of cards, and both require operations like adding and removing cards.

> 继承就是基于已有的类进行修改来获取新类的能力。举个例子，比方说我们需要一个表示『一手牌』的类，这个就是指一个牌手手中拿着的牌。『一手牌』和『一副牌』有些相似：都是由一系列的纸牌组成的，也都要有添加和移除纸牌的运算。

A hand is also different from a deck; there are operations we want for hands that don’t make sense for a deck. For example, in poker we might compare two hands to see which one wins. In bridge, we might compute a score for a hand in order to make a bid.

> 『一手牌』还和『一副牌』有所区别；对于手中的牌有一些运算并不适用于整副的牌。比如说，在扑克游戏中，我们可能需要对比两手牌来看看哪一副胜利。在桥牌里面，还可能需要对手中的牌进行计分以决胜负。

This relationship between classes—similar, but different—lends itself to inheritance. To define a new class that inherits from an existing class, you put the name of the existing class in parentheses:

> 类之间这种相似又有区别的关系，就适合用继承来实现了。要继承一个已有的类来定义新类，就要把已有类的名字放到括号中，如下所示：

```
class Hand(Deck):
"""Represents a hand of playing cards."""
```

This definition indicates that Hand inherits from Deck; that means we can use methods like pop_card and add_card for Hands as well as Decks.

> 上面这样的定义就表示了 Hand 继承了 Deck；也就意味着我们可以在 Hands 中使用 Decks 中的那些方法，比如 pop_card 以及 add_card 等等。

When a new class inherits from an existing one, the existing one is called the parent and the new class is called the child.

> 当一个新类继承了一个已有的类时，这个已有的类就叫做基类，新定义的类叫做子类。

In this example, Hand inherits __init__ from Deck, but it doesn't really do what we want: instead of populating the hand with 52 new cards, the init method for Hands should initialize cards with an empty list. If we provide an init method in the Hand class, it overrides the one in the Deckclass:

> 在本章的这个例子中，Hand 类从 Deck 类继承了__init__方法，但这个方法和我们的需求还不一样：Hand类的 init 方法应该用一个空列表来初始化手中的牌，而不是像 Deck 类中那样用一整副52张牌。

```
# inside class Hand:
def __init__(self, label=''):
    self.cards = []
    self.label = label
```

When you create a Hand, Python invokes this init method, not the one in Deck.

> 像上面这样改写一下之后，这样再建立一个 Hand 类的时候，Python 就会调用这个自定义的 init 方法，而不是 Deck 当中的。

```
>>> hand = Hand('new hand')
>>> hand = Hand('new hand')
>>> hand.cards []
>>> hand.cards []
>>> hand.label
>>> hand.label
'new hand'
```

The other methods are inherited from Deck, so we can use pop_card and add_card to deal a card:

> 其他方法都从 Deck 类中继承了过来，所以我们就可以直接用 pop_card 和 add_card 方法来处理纸牌了：

```
>>> deck = Deck()
>>> deck = Deck()
>>> card = deck.pop_card()
>>> card = deck.pop_card()
>>> hand.add_card(card)
>>> hand.add_card(card)
>>> print(hand)
>>> print(hand)
King of Spades
```

A natural next step is to encapsulate this code in a method called move_cards:

> 接下来很自然地，我们把这段名为 move_cards 的方法放进去：

```
#inside class Deck:
def move_cards(self, hand, num):
    for i in range(num):
        hand.add_card(self.pop_card())
```

move_cards takes two arguments, a Hand object and the number of cards to deal. It modifies both self and hand, and returns None.

> move_cards 方法接收两个参数，一个 Hand 对象，以及一个要处理的纸牌数量。该方法会修改 self 和 hand。返回为空。

In some games, cards are moved from one hand to another, or from a hand back to the deck. You can use move_cards for any of these operations: self can be either a Deck or a Hand, and hand, despite the name, can also be a Deck.

> 在有的游戏中，纸牌需要从一手牌拿出去放到另外一手牌中去，或者从手中拿出去放到牌堆里面。这就亏用 move_cards 来实现这些操作：第一个变量 self 可以是一副牌也可以是一手牌，第二个变量虽然名字是 hand，实际上也可以是一个 Deck 对象。

Inheritance is a useful feature. Some programs that would be repetitive without inheritance can be written more elegantly with it. Inheritance can facilitate code reuse, since you can customize the behavior of parent classes without having to modify them. In some cases, the inheritance structure reflects the natural structure of the problem, which makes the design easier to understand.

> 继承是一个很有用的功能。有的程序如果不用继承的话就会有很多重复代码，用继承来写出来就会更简洁很多了。继承有助于代码重用，因为你可以对基类的行为进行定制而不用去修改基类本身。在某些情况下，继承的结构也反映了要解决的问题中的自然关系，这就让程序设计更易于理解。

On the other hand, inheritance can make programs difficult to read. When a method is invoked, it is sometimes not clear where to find its definition. The relevant code may be spread across several modules. Also, many of the things that can be done using inheritance can be done as well or better without it.

> 然而继承也容易降低程序可读性。当调用一个方法的时候，有时候不容易找到该方法的定义位置。相关的代码可能跨了好几个模块。此外，很多事情可以用继承来实现，但不用继承也能做到同样效果，甚至做得更好。

18.8 Class diagrams 类图

So far we have seen stack diagrams, which show the state of a program, and object diagrams, which show the attributes of an object and their values. These diagrams represent a snapshot in the execution of a program, so they change as the program runs.

> 目前为止，我们见过栈图了，栈图是展示一个程序的状态的，我们还见过对象图了，表示的是一个对象中的各个属性及其值。这些图都是对一个程序运行中某个瞬间的反映，因此随着程序运行而产生变化。

They are also highly detailed; for some purposes, too detailed. A class diagram is a more abstract representation of the structure of a program. Instead of showing individual objects, it shows classes and the relationships between them.

> 这些图解还都非常详细；有的时候就都过于繁琐冗余了。而类图则是对一个程序结构的更抽象的表示。类图并不会表现出各个独立的对象，而是会表现出程序中的各个类以及它们之间的关系。

There are several kinds of relationship between classes:

> 类之间有很多种关系，大概如下所示：

• Objects in one class might contain references to objects in another class. For example, each Rectangle contains a reference to a Point, and each Deck contains references to many Cards. This kind of relationship is called HAS-A, as in, “a Rectangle has a Point.”

> 一个类中的对象可能包含了另一个类对象的引用。例如，每一个 Rectangle （矩形）对象都包含了对 Point（点）的引用，而每一个 Deck （成副的牌）对象都包含了对很多个 Card （纸牌）对象的引用。这种关系也叫做『含有』，就好比是说，『一个矩形中含有一个点。』

• One class might inherit from another. This relationship is called IS-A, as in, “a Hand is a kind of a Deck.”

> 一类可能继承了其他的类。这种关系也可以叫做『是一个』，比如说，『一手牌就是一种牌的组合。』

• One class might depend on another in the sense that objects in one class take objects in the second class as parameters, or use objects in the second class as part of a computation. This kind of relationship is called a dependency.

> 一种类可能要依赖其他类，比如一个类中的对象用另外一个类中的对象作为参数，或者用做计算中的某一部分。这种关系就叫做『依赖』。

A class diagram is a graphical representation of these relationships. For example, Figure 18.2 shows the relationships between Card, Deck and Hand.

> 类图就是对这些关系的一个图形化的表示。比如，在途18.2中，就展示了 Card，Deck 以及 Hand 三个类的关系。

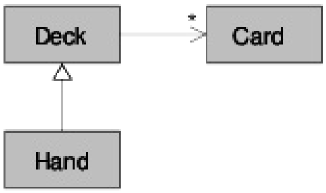


Figure 18.2: Class diagram.

The arrow with a hollow triangle head represents an IS-A relationship; in this case it indicates that Hand inherits from Deck.

> 有空心三角形的箭头表示了『是一个』的关系；在这里意思就是 Hand 继承了 Deck。

The standard arrow head represents a HAS-A relationship; in this case a Deck has references to Card objects.

> 另一个箭头表示了『有一个』的关系；在这里的意思是 Deck 当中有若干对 Card 对象的引用。

The star (*) near the arrow head is a multiplicity; it indicates how many Cards a Deck has. A multiplicity can be a simple number, like 52, a range, like 5..7 or a star, which indicates that a Deck can have any number of Cards.

> 箭头处有个小星号*；这里可以表明一个 Deck 中含有的 Card的个数。可以标出个数，比如52，或者是范围，比如5..7或者一个星号，这就意味着一个 Deck 中可以含有任意个数的 Card。

There are no dependencies in this diagram. They would normally be shown with a dashed arrow. Or if there are a lot of dependencies, they are sometimes omitted.

> 这个图解中没有出现依赖关系。这种关系一般用虚线箭头来表示。或者当依赖关系很多的时候，有时候就都忽略掉了。

A more detailed diagram might show that a Deck actually contains a list of Cards, but built-in types like list and dict are usually not included in class diagrams.

> 更细节化的图解就可能表现出一个 Deck 中会包含一个 Card 对象组成的列表，但一般情况下类图不会包括内置类型比如列表和字典。

18.9 Debugging 调试

Inheritance can make debugging difficult because when you invoke a method on an object, it might be hard to figure out which method will be invoked.

> 继承可以让调试变得很夸你呢，因为你调用某个对象中的某个方法的时候，很难确定到底是调用的哪一个方法。

Suppose you are writing a function that works with Hand objects. You would like it to work with all kinds of Hands, like PokerHands, BridgeHands, etc. If you invoke a method like shuffle, you might get the one defined in Deck, but if any of the subclasses override this method, you'll get that version instead. This behavior is usually a good thing, but it can be confusing.

> 假设你写一个处理 Hand 对象的函数。你可能要让该函数适用于所有类型的牌型，比如常规牌型，桥牌牌型等等。假设你要调用洗牌的方法 shuffle，你可能用的是 Deck 类当中的，不过如果子类当中有覆盖的该方法，你运行的就是子类中的方法了。这种行为一般是很有好处的，不过也容易把人弄糊涂。

Any time you are unsure about the flow of execution through your program, the simplest solution is to add print statements at the beginning of the relevant methods. If Deck.shuffle prints a message that says something like Running Deck.shuffle, then as the program runs it traces the flow of execution.

> 在你的程序运行的过程中，只要你对程序流程有疑问了，就可以在相关的方法头部添加 print 语句来打印输出一下信息，这就是最简单的解决方法了。如果 Deck.shuffle 输出了信息比如说『在运行 Deck 的 shuffle』，那就可以根据这些信息来追踪执行流程了。

As an alternative, you could use this function, which takes an object and a method name (as a string) and returns the class that provides the definition of the method:

> 另外一个思路，就是用下面这个函数，该函数接收一个对象和一个方法的名字（作为字符串），然后返回提供该方法定义的类的名称。

```
def find_defining_class(obj, meth_name):
    for ty in type(obj).mro():
        if meth_name in ty.__dict__:
            return ty
```

Here’s an example:

> 如下所示：

```
>>> hand = Hand()
>>> hand = Hand()
>>> find_defining_class(hand, 'shuffle')
>>> find_defining_class(hand, 'shuffle')
<class 'Card.Deck'>
```

So the shuffle method for this Hand is the one in Deck. find_defining_class uses the mro method to get the list of class objects (types) that will be searched for methods. “MRO” stands for “method resolution order”, which is the sequence of classes Python searches to “resolve” a method name.

> 所这样就能判断这里面 Hand 中的 shuffle 方法是来自 Deck 的。
>
> find_defining_class 用了 mro 方法来获取所有搜索方法的类对象的列表。
>
> 『MRO』的意思是『method resolution order（方法 解决方案 顺序）』， 也就是 Python 搜索来找到方法名的类的序列。

Here’s a design suggestion: when you override a method, the interface of the new method should be the same as the old. It should take the same parameters, return the same type, and obey the same preconditions and postconditions. If you follow this rule, you will find that any function designed to work with an instance of a parent class, like a Deck, will also work with instances of child classes like a Hand and PokerHand.

> 下面是一个在设计上的建议：当你覆盖一个方法的时候，新的方法的接口最好同旧的完全一致。应该接收同样的参数，返回同样类型，并且遵循同样的前置条件和后置条件。只要你遵守这个规则，你就会发现所有之前设计来处理一个基类的函数，比如处理 Deck 的，就都可以用于子类的实例上面，比如 Hand 类或者 PokerHand 类。

If you violate this rule, which is called the “Liskov substitution principle”, your code will collapse like (sorry) a house of cards.

> 如果你违背了上面这个『里氏替换原则』， 你的代码就可能很悲剧地崩溃，就像无数纸牌坍塌一样。

18.10 Data encapsulation 数据封装

The previous chapters demonstrate a development plan we might call “object-oriented design”. We identified objects we needed—like Point,Rectangle and Time—and defined classes to represent them. In each case there is an obvious correspondence between the object and some entity in the real world (or at least a mathematical world).

> 之前的章节中，我们展示了所谓『面向对象设计』的开发规划模式。在这些章节中，我们显示确定好需要的对象——比如点，矩形以及时间——然后再定义一些类去代表这些内容。在这些例子中，类的对象与现实世界（或者至少是数学世界）中的一些实体都有显著的对应关系。

But sometimes it is less obvious what objects you need and how they should interact. In that case you need a different development plan. In the same way that we discovered function interfaces by encapsulation and generalization, we can discover class interfaces by data encapsulation.

> 不过有时候就不太好确定具体需要什么样的对象，以及如何去实现。这时候就需要一种完全不同的开发规划模式了。之前我们对函数接口进行过封装和泛化的处理，现在也可以通过数据封装来改进类的接口。

Markov analysis, from Section 13.8, provides a good example. If you download my code from Here , you'll see that it uses two global variables—suffix_map and prefix—that are read and written from several functions.

> 比如马科夫分析，在13.8中出现的，就是一个很好的例子。如果你从这里下载了我的样例代码，你就会发现这里用了两个全局变量—suffix_map 以及 prefix—这两个全局变量会被多个函数读取和写入。

```
suffix_map = {}
prefix = ()
```

Because these variables are global, we can only run one analysis at a time. If we read two texts, their prefixes and suffixes would be added to the same data structures (which makes for some interesting generated text).

> 这些变量是全局的，因此我们每次只运行了一次分析。如果我们要读取两个文本，他们的前置和后置词汇都会被添加到同样的数据结构上面去（这样就能生成一些有趣的机器制造的文本了）。

To run multiple analyses, and keep them separate, we can encapsulate the state of each analysis in an object. Here's what that looks like:

> 如果要运行多次分析，并且要对这些分析进行区分，我们可以把每次分析的状态封装到对象中。如下所示：

```
class Markov:
    def __init__(self):
        self.suffix_map = {}
        self.prefix = ()
```

Next, we transform the functions into methods. For example, here's process_word:

> 接下来就是把各个函数转换成方法。例如下面就是process_word 方法：

```
def process_word(self, word, order=2):
    if len(self.prefix) < order:
        self.prefix += (word,)
    return
    try:
        self.suffix_map[self.prefix].append(word)
    except
        KeyError:             # if there is no entry for this prefix, make one
    self.suffix_map[self.prefix] = [word]
    self.prefix = shift(self.prefix, word)
```

Transforming a program like this—changing the design without changing the behavior—is another example of refactoring (see Section 4.7).

> 上面这种方式对程序进行的修改只是改变了设计，而不改变程序的行为，这就是重构的另一个例子（参考4.7）。

This example suggests a development plan for designing objects and methods:

> 这一样例展示了一种设计累的对象和方法的开发规划模式：

1. Start by writing functions that read and write global variables (when necessary).

> 先开始写一些函数来读去和写入全局变量（在必要的情况下）。

1. Once you get the program working, look for associations between global variables and the functions that use them.

> 一旦程序可以工作了，就检查一下全局变量与使用它们的函数之间的关系。

1. Encapsulate related variables as attributes of an object.

> 把相关的变量作为类的属性封装到一起。

1. Transform the associated functions into methods of the new class.

> 把相关的函数转换成新类的方法。

As an exercise, download my Markov code from Here, and follow the steps described above to encapsulate the global variables as attributes of a new class called Markov. Solution (note the capital M).

> 做一个练习，从这里下载我的马科夫分析代码，然后根据上面说的步骤来一步步把全局变量封装成一个名为 Markov 的新类的属性。样例代码 (一定要注意 M 是大写的哈)

18.11 Glossary 术语列表

encode: To represent one set of values using another set of values by constructing a mapping between them.

> 编码：通过建立映射的方式来用一系列的值来表示另外一系列的值。

class attribute: An attribute associated with a class object. Class attributes are defined inside a class definition but outside any method.

> 类的属性：属于某个类的对象的属性。类的属性都在类定义的内部，在类内方法之外。

instance attribute: An attribute associated with an instance of a class.

> 实例属性：属于某个类的实例的属性。

veneer: A method or function that provides a different interface to another function without doing much computation.

> 嵌板：某一方法或者函数，为另外的函数提供了不同的界面，而没有做额外运算。

inheritance: The ability to define a new class that is a modified version of a previously defined class.

> 继承：基于已定义过的类，进行修改来定义一个新类，这种特性就是继承。

parent class: The class from which a child class inherits.

> 基类：被子类继承的类。

child class: A new class created by inheriting from an existing class; also called a “subclass”.

> 子类：基于已有类而建立的新类；也称为『分支类』。

IS-A relationship: A relationship between a child class and its parent class.

> 『是一个』关系：一种子类与基类之间的关系。

HAS-A relationship: A relationship between two classes where instances of one class contain references to instances of the other.

> 『有一个』关系：某一个类的实例中包含其他类的实例的引用的关系。

dependency: A relationship between two classes where instances of one class use instances of the other class, but do not store them as attributes.

> 依赖关系：两个类之间的一种关系，一个类的实例使用了另外一个类的实例，但并未作为属性来存储。

class diagram: A diagram that shows the classes in a program and the relationships between them.

> 类图：一种展示程序中各个类及其之间关系的图解。

multiplicity: A notation in a class diagram that shows, for a HAS-A relationship, how many references there are to instances of another class.

> 多样性：类图中显示的一种记号，适用于『有一个』关系中，表示一个类当中另一个类的实例的引用的个数。

data encapsulation: A program development plan that involves a prototype using global variables and a final version that makes the global variables into instance attributes.

> 数据封装：一种程序开发规划方式，用全局变量做原型体，然后逐步将这些全局变量转换成实例的属性。

18.12 Exercises 练习

Exercise 1 练习1

For the following program, draw a UML class diagram that shows these classes and the relationships among them.

> 阅读下面的代码，画一个 UML 类图，表示出程序中的类，以及类之间的关系。

```
class PingPongParent:
    pass  class Ping(PingPongParent):
    def __init__(self, pong):
            self.pong = pong
class Pong(PingPongParent):
    def __init__(self, pings=None):
        if pings is None:
            self.pings = []
        else:
            self.pings = pings
    def add_ping(self, ping):
        self.pings.append(ping)
        pong = Pong()
        ping = Ping(pong)
        pong.add_ping(ping)
```

Exercise 2 练习2

Write a Deck method called deal_hands that takes two parameters, the number of hands and the number of cards per hand. It should create the appropriate number of Hand objects, deal the appropriate number of cards per hand, and return a list of Hands.

> 为 Deck 类写一个名为 deal_hands 的方法，接收两个参数，一个为牌型数量，一个为每一个牌型的纸牌数。该方法需要创建适当的牌型对象数量，处理适当的每个牌型中的纸牌数，然后返回一个牌型组成的列表。

Exercise 3 练习3

The following are the possible hands in poker, in increasing order of value and decreasing order of probability:

> 下面是扑克牌中可能的各个牌型，排列顺序为值的升序，出现概率的降序：

pair: two cards with the same rank

> 一对：两张同样牌值的牌

two pair: two pairs of cards with the same rank

> 双对：两对同样牌值的牌

three of a kind: three cards with the same rank

> 三张：三张同样牌值的牌

straight: five cards with ranks in sequence (aces can be high or low, so Ace-2-3-4-5 is a straight and so is 10-Jack-Queen-King-Ace, but Queen-King-Ace-2-3 is not.)

> 顺子：五张牌值连续的牌（A 可以用作开头，也可以用作结尾，所以 A-2-3-4-5是一个顺子，10-J-Q-K-A 也是一个，但 Q-K-A-2-3就不行了。）

flush: five cards with the same suit

> 同花：五张牌花色一致

full house: three cards with one rank, two cards with another

> 三带二：三张同牌值的牌，两张另外的同牌值的牌

four of a kind: four cards with the same rank

> 四条：四张同一牌值的牌

straight flush: five cards in sequence (as defined above) and with the same suit

> 同花顺：五张组成顺子并且是同一花色的牌

The goal of these exercises is to estimate the probability of drawing these various hands.

> 此次练习的目的就是要估计获得以上各个牌型的概率。

1. Download the following files from Here

> 从这个网址下载下面的文件：

:Card.py : A complete version of the Card, Deck and Hand classes in this chapter.

> 该文件是本章所涉及的 Card，Deck 以及 Hand 类的完整实现。

PokerHand.py : An incomplete implementation of a class that represents a poker hand, and some code that tests it.

> 该文件是一个不完整版本的类，表示的是一个牌型，以及一些测试代码。

1. If you run PokerHand.py, it deals seven 7-card poker hands and checks to see if any of them contains a flush. Read this code carefully before you go on.

> 如果你运行 PokerHand.py，改程序会处理七个七张牌的牌型，然后检查是否其中包含一副顺子。
>
> 好好阅读一下这份代码，然后再继续后面的练习。

1. Add methods to PokerHand.py named has_pair, has_twopair, etc. that return True or False according to whether or not the hand meets the relevant criteria. Your code should work correctly for “hands” that contain any number of cards (although 5 and 7 are the most common sizes).

> 在 PokerHand.py 里面增加名为has_pair, has_twopair等等方法。这些方法根据牌型中是否满足特定的组合而返回 True 或者 False。你的代码应该能适用于有任意张牌的牌型（虽然5或者7是最常见的牌数）。

1. Write a method named classify that figures out the highest-value classification for a hand and sets the label attribute accordingly. For example, a 7-card hand might contain a flush and a pair; it should be labeled “flush”.

> 写一个名为 classify 的函数，判断出一副牌型中的最高值的一份，然后用来命名到标签属性。例如，一个七张牌的牌型可能包含一个顺子和一个对子；这就应该被标为『顺子』。

1. When you are convinced that your classification methods are working, the next step is to estimate the probabilities of the various hands. Write a function in PokerHand.py that shuffles a deck of cards, divides it into hands, classifies the hands, and counts the number of times various classifications appear.

> 当你确定你的分类方法运转正常了，下一步就是要估计各个牌型的出现概率。在 PokerHand.py 中写一个函数来对一副牌进行洗牌，分成多个牌型，对各个牌型进行分类，然后统计不同类型出现的次数。

1. Print a table of the classifications and their probabilities. Run your program with larger and larger numbers of hands until the output values converge to a reasonable degree of accuracy. Compare your results to the values at Here.

> 打印输出一个由类型和概率组成的列表。逐步用大规模的牌型来测试你的程序，直到输出的值趋向于一个比较合理的准确范围。把你的运行结果与这里的结果进行对比。

Solution 样例代码

Chapter 19 The Goodies 额外补充

One of my goals for this book has been to teach you as little Python as possible. When there were two ways to do something, I picked one and avoided mentioning the other. Or sometimes I put the second one into an exercise.

> 我在本书中的一个目标就是尽量少教你 Python（译者注：而要多教编程）。有的时候完成一个目的有两种方法，我都会只选择一种而不提其他的。或者有的时候我就把第二个方法放到练习里面。

Now I want to go back for some of the good bits that got left behind. Python provides a number of features that are not really necessary—you can write good code without them—but with them you can sometimes write code that's more concise, readable or efficient, and sometimes all three.

> 现在我就要往回倒车一下，捡起一些当时略过的重要内容来给大家讲一下。Python 提供了很多并非必须的功能—你完全可以不用这些功能也能写出很好的代码—但用这些功能有时候能让你的代码更加简洁，可读性更强，或者更有效率，甚至有时候能兼顾这三个方面。

19.1 Conditional expressions 条件表达式

We saw conditional statements in Section 5.4. Conditional statements are often used to choose one of two values; for example:

> 在5.4中，我们见到了条件语句。条件语句往往用于二选一的情况下；比如：

```
if x > 0:
    y = math.log(x)
else:
    y = float('nan')
```

This statement checks whether x is positive. If so, it computes math.log. If not,math.log would raise a ValueError. To avoid stopping the program, we generate a "NaN", which is a special floating-point value that represents "Not a Number".

> 这个语句检查了 x 是否为正数。如果为正数，程序就计算对数值 math.log。如果非正，对数函数会返回一个值错误 ValueError。要避免因此而导致程序异常退出，咱们就生成了一个『NaN』，这个符号是一个特别的浮点数的值，表示的意思是『不是一个数』。

We can write this statement more concisely using a conditional expression:

> 用一个条件表达式能让这个语句更简洁：

```
y = math.log(x)     if x > 0     else     float('nan')
```

You can almost read this line like English: “y gets log-x if x is greater than 0; otherwise it gets NaN”. Recursive functions can sometimes be rewritten using conditional expressions. For example, here is a recursive version of factorial:

> 上面这行代码读起来就跟英语一样了：『如果 x 大于0就让 y 等于 x 的对数；否则的话就返回 Nan』。
>
> 递归函数有时候也可以用这种条件表达式来改写。例如下面就是分形函数 factorial 的一个递归版本：

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

We can rewrite it like this:

> 我们可以这样改写：

```
def factorial(n):
    return 1 if n == 0  else  return n * factorial(n-1)
```

Another use of conditional expressions is handling optional arguments. For example, here is the init method from GoodKangaroo (see Exercise 2):

> 条件表达式还可以用于处理可选参数。例如下面就是练习2中 GoodKangaroo 类的 init 方法：

```
def __init__(self, name, contents=None):
    self.name = name
    if contents == None:
        contents = []
    self.pouch_contents = contents
```

We can rewrite this one like this:

> 我们可以这样来改写：

```
def __init__(self, name, contents=None):
    self.name = name
    self.pouch_contents = []
    if contents == None else contents
```

In general, you can replace a conditional statement with a conditional expression if both branches contain simple expressions that are either returned or assigned to the same variable.

> 一般来讲，你可以用条件表达式来替换掉条件语句，无论这些语句的分支是返回语句或者是赋值语句。

19.2 List comprehensions 列表推导

In Section 10.7 we saw the map and filter patterns. For example, this function takes a list of strings, maps the string method capitalize to the elements, and returns a new list of strings:

> 在10.7当中，我们看到了映射和过滤模式。例如，下面这个函数接收一个字符串列表，然后将每一个元素都用字符串方法 capitalize 处理成大写的，然后返回一个新的字符串列表：

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

We can write this more concisely using a list comprehension:

> 用列表推导就可以将上面的代码写得更简洁：

```
def capitalize_all(t):
    return [s.capitalize() for s in t]
```

The bracket operators indicate that we are constructing a new list. The expression inside the brackets specifies the elements of the list, and the for clause indicates what sequence we are traversing.

> 方括号的意思是我们正在建立一个新的列表。方括号内的表达式确定了此新列表中的元素，然后 for 语句表明我们要遍历的序列。

The syntax of a list comprehension is a little awkward because the loop variable, s in this example, appears in the expression before we get to the definition.

> 列表推导的语法有点复杂，就因为这个循环变量，在上面例子中是 s，这个 s 在我们定义之前就出现在语句中了。

List comprehensions can also be used for filtering. For example, this function selects only the elements of t that are upper case, and returns a new list:

> 列表推导也可以用到滤波中。例如，下面的函数从 t 中选择了大写的元素，然后返回成一个新的列表：

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

We can rewrite it using a list comprehension:

> 咱们可以用列表推导来重写这个函数：

```
def only_upper(t):
    return [s for s in t if s.isupper()]
```

List comprehensions are concise and easy to read, at least for simple expressions. And they are usually faster than the equivalent for loops, sometimes much faster. So if you are mad at me for not mentioning them earlier, I understand.

> 列表推导很简洁，也很容易阅读，至少在简单的表达式上是这样。这些语句的执行也往往比同样情况下的 for 循环更快一些，有时候甚至快很多。所以如果你因为我没有早些给你讲而发怒，我也能理解。

But, in my defense, list comprehensions are harder to debug because you can't put a print statement inside the loop. I suggest that you use them only if the computation is simple enough that you are likely to get it right the first time. And for beginners that means never.

> 但是，我也要辩护一下，列表推导会导致调试非常困难，因为你不能在循环内部放 print 语句了。我建议你只去在一些简单的地方使用，要确保你第一次写出来就能保证代码正常工作。也就是说初学者就还是别用为好。

19.3 Generator expressions 生成器表达式

Generator expressions are similar to list comprehensions, but with parentheses instead of square brackets:

> 生成器表达式与列表推导相似，用的不是方括号，而是圆括号：

```
>>> g = (x**2 for x in range(5))
>>> g = (x**2 for x in range(5))
>>> g
>>> g
<generator object <genexpr> at 0x7f4c45a786c0>
```

The result is a generator object that knows how to iterate through a sequence of values. But unlike a list comprehension, it does not compute the values all at once; it waits to be asked. The built-in function next gets the next value from the generator:

> 上面这样运行得到的结果就是一个生成器对象，用来遍历一个值的序列。但与列表推导不同的是，生成器表达式并不会立即计算出所有的值；它要等着被调用。内置函数 next 会从生成器中得到下一个值：

```
>>> next(g)
>>> next(g)
0
>>> next(g)
>>> next(g)
1
```

When you get to the end of the sequence, next raises a StopIteration exception. You can also use a for loop to iterate through the values:

> 当程序运行到序列末尾的时候，next 函数就会抛出一个停止遍历的异常。你也可以用一个 for 循环来遍历所有的值：

```
>>> for val in g: ...
>>> for val in g: ...
print(val)
4
9
16
```

The generator object keeps track of where it is in the sequence, so the for loop picks up where next left off. Once the generator is exhausted, it continues to raise StopException:

> 生成器对象能够追踪在序列中的位置，所以 for 语句就会在 next 函数退出的地方开始。一旦生成器使用完毕了，接下来就要抛出一个停止异常了：

```
>>> next(g)
>>> next(g)
StopIteration
```

Generator expressions are often used with functions like sum, max, and min:

> 生成器表达式多用于求和、求最大或者最小这样的函数中：

```
>>> sum(x**2 for x in range(5))
>>> sum(x**2 for x in range(5))
30
```

19.4 any and all 任意和所有

Python provides a built-in function, any, that takes a sequence of boolean values and returns True if any of the values are True. It works on lists:

> Python 提供了一个名为 any 的内置函数，该函数接收一个布尔值序列，只要里面有任意一个是真，就返回真。该函数适用于列表：

```
>>> any([False, False, True])
>>> any([False, False, True])
True
```

But it is often used with generator expressions:

> 但这个函数多用于生成器表达式中：

```
>>> any(letter == 't' for letter in 'monty')
>>> any(letter == 't' for letter in 'monty')
True
```

That example isn't very useful because it does the same thing as the in operator. But we could use any to rewrite some of the search functions we wrote in Section 9.3. For example, we could write avoids like this:

> 这个例子没多大用，因为效果和 in 运算符是一样的。但我们能用 any 函数来改写我们在9.3中写的一些搜索函数。例如，我们可以用如下的方式来改写 avoids：

```
def avoids(word, forbidden):
    return not any(letter in forbidden for letter in word)
```

The function almost reads like English, "word avoids forbidden if there are not any forbidden letters in word."

> 这样这个函数读起来基本就跟英语一样了。

Using any with a generator expression is efficient because it stops immediately if it finds a True value, so it doesn't have to evaluate the whole sequence.

> 用 any 函数和生成器表达式来配合会很有效率，因为只要发现真值程序就会停止了，所以并不需要对整个序列进行运算。

Python provides another built-in function, all, that returns True if every element of the sequence is True. As an exercise, use all to re-write uses_all from Section 9.3.

> Python 还提供了另外一个内置函数 all，该函数在整个序列都是真的情况下才返回真。
>
> 做个练习，用 all 来改写一下9.3中的uses_all 函数。

19.5 Sets 集合

In Section 13.6 I use dictionaries to find the words that appear in a document but not in a word list. The function I wrote takes d1, which contains the words from the document as keys, and d2, which contains the list of words. It returns a dictionary that contains the keys from d1 that are not in d2.

> 在13.6中，我用了字典来查找存在于文档中而不存在于词汇列表中的词汇。我写的这个函数接收两个参数，一个是 d1是包含了文档中的词作为键，另外一个是 d2包含了词汇列表。程序会返回一个字典，这个字典包含的键存在于 d1而不在 d2中。

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

In all of these dictionaries, the values are None because we never use them. As a result, we waste some storage space.

> 在这些字典中，键值都是 None，因为根本没有使用。结果就是，浪费了一些存储空间。

Python provides another built-in type, called a set, that behaves like a collection of dictionary keys with no values. Adding elements to a set is fast; so is checking membership. And sets provide methods and operators to compute common set operations.

> Python 还提供了另一个内置类型，名为 set（也就是集合的意思），其性质就是有字典的键而无键值。
>
> 对集合中添加元素是很快的；对集合成员进行检查也很快。此外集合还提供了一些方法和运算符来进行常见的集合运算。

For example, set subtraction is available as a method called difference or as an operator, -. So we can rewrite subtract like this:

> 例如，集合的减法就可以用一个名为 difference 的方法，或者就用减号-。所以我们可以把 subtract 改写成如下形式：

```
def subtract(d1, d2):
    return set(d1) - set(d2)
```

The result is a set instead of a dictionary, but for operations like iteration, the behavior is the same.

> 上面这个函数的结果就是一个集合而不是一个字典，但对于遍历等等运算来说，用起来都一样的。

Some of the exercises in this book can be done concisely and efficiently with sets. For example, here is a solution to has_duplicates, from Exercise 7, that uses a dictionary:

> 本书中的一些练习都可以通过使用集合而改写成更精简更高效的形式。例如，下面的代码就是 has_duplicates 的一个实现方案，来自练习7，用的是字典：

```
def has_duplicates(t):
    d = {}
    for x in t:
        if x in d:
            return True
    d[x] = True
    return False
```

When an element appears for the first time, it is added to the dictionary. If the same element appears again, the function returns True.

> 当一个元素第一次出现的时候，就被添加到字典中。如果同一个元素又出现了，该函数就返回真。

Using sets, we can write the same function like this:

> 用集合的话，我们就能把该函数写成如下形式：

```
def has_duplicates(t):
    return len(set(t)) < len(t)
```

An element can only appear in a set once, so if an element in t appears more than once, the set will be smaller than t. If there are no duplicates, the set will be the same size as t.

> 一个元素在一个集合中只能出现一次，所以如果一个元素在 t 中出现次数超过一次，集合会比 t 规模小一些。如果没有重复，集合的规模就应该和 t 一样大。

We can also use sets to do some of the exercises in Chapter 9. For example, here's a version of uses_only with a loop:

> 我们还能用集合来做一些第九章的练习。例如，下面就是用一个循环实现的一个版本的 uses_only：

```
def uses_only(word, available):
    for letter in word:
    if letter not in available:
        return False
    return True
```

uses_only checks whether all letters in word are in available. We can rewrite it like this:

> uses_only 会检查 word 中的所有字母是否出现在 available 中。我们可以用如下方法重写：

```
def uses_only(word, available):
    return set(word) <= set(available)
```

The <= operator checks whether one set is a subset or another, including the possibility that they are equal, which is true if all the letters in word appear in available. As an exercise, rewrite avoids using sets.

> 这里的<=运算符会检查一个集合是否切另外一个集合的子集或者相等，如果 word 中所有的字符都出现在 available 中就返回真。

19.6 Counters 计数器

A Counter is like a set, except that if an element appears more than once, the Counter keeps track of how many times it appears. If you are familiar with the mathematical idea of a multiset, a Counter is a natural way to represent a multiset.

> 计数器跟集合相似，除了一点，就是如果计数器中元素出现的次数超过一次，计数器会记录下出现的次数。如果你对数学上多重集的概念有所了解，就会知道计数器是一种对多重集的表示方式。

Counter is defined in a standard module called collections, so you have to import it. You can initialize a Counter with a string, list, or anything else that supports iteration:

> 计数器定义在一个名为 collections 的标准模块中，所以你必须先导入一下。你可以用字符串，列表或者任何支持遍历的类型来初始化一个计数器：

```
>>> from collections import Counter
>>> from collections import Counter
>>> count = Counter('parrot')
>>> count = Counter('parrot')
>>> count
>>> count
Counter({'r': 2, 't': 1, 'o': 1, 'p': 1, 'a': 1})
```

Counters behave like dictionaries in many ways; they map from each key to the number of times it appears. As in dictionaries, the keys have to be hashable.

> 计数器的用法与字典在很多方面都相似；二者都映射了每个键到出现的次数上。在字典中，键必须是散列的。

Unlike dictionaries, Counters don't raise an exception if you access an element that doesn't appear. Instead, they return 0:

> 与字典不同的是，当你读取一个不存在的元素的时候，计数器并不会抛出异常。相反的，这时候程序会返回0：

```
>>> count['d']
>>> count['d']
0
```

We can use Counters to rewrite is_anagram from Exercise 6:

> 我们可以用计数器来重写一下练习6中的这个 is_anagram 函数：

```
def is_anagram(word1, word2):
    return Counter(word1) == Counter(word2)
```

If two words are anagrams, they contain the same letters with the same counts, so their Counters are equivalent. Counters provide methods and operators to perform set-like operations, including addition, subtraction, union and intersection. And they provide an often-useful method, most_common, which returns a list of value-frequency pairs, sorted from most common to least:

> 如果两个单词是换位词，他们包含同样个数的同样字母，所以他们的计数器是相等的。
>
> 计数器提供了一些方法和运算器来运行类似集合的运算，包括加法，剪发，合并和交集等等。此外还提供了一个最常用的方法，most_common，该方法会返回一个由值-出现概率组成的数据对的列表，按照概率从高到低排列：

```
>>> count = Counter('parrot')
>>> count = Counter('parrot')
>>> for val, freq in count.most_common(3): ...
>>> for val, freq in count.most_common(3): ...
print(val, freq)
r 2
p 1
a 1
```

19.7 defaultdict 默认字典

The collections module also provides defaultdict, which is like a dictionary except that if you access a key that doesn't exist, it can generate a new value on the fly.

> collection 模块还提供了一个默认字典，与普通字典的区别在于当你读取一个不存在的键的时候，程序会添加上一个新值给这个键。

When you create a defaultdict, you provide a function that's used to create new values. A function used to create objects is sometimes called a factory. The built-in functions that create lists, sets, and other types can be used as factories:

> 当你创建一个默认字典的时候，就提供了一个能创建新值的函数。用来创建新对象的函数也被叫做工厂。内置的创建列表、集合以及其他类型的函数都可以被用作工厂：

```
>>> from collections import defaultdict
>>> from collections import defaultdict
>>> d = defaultdict(list)
>>> d = defaultdict(list)
```

Notice that the argument is list, which is a class object, not list(), which is a new list. The function you provide doesn't get called unless you access a key that doesn't exist.

> 要注意到这里的参数是一个列表，是一个类的对象，而不是 list()，带括号的就是一个新列表了。这个创建新值的函数只有当你试图读取一个不存在的键的时候才会被调用。

```
>>> t = d['new key']
>>> t = d['new key']
>>> t []
>>> t []
```

The new list, which we're calling t, is also added to the dictionary. So if we modify t, the change appears in d:

> 新的这个我们称之为 t 的列表，也会被添加到字典中。所以如果我们修改 t，这种修改也会在 d 中出现。

```
>>> t.append('new value')
>>> t.append('new value')
>>> d
>>> d
defaultdict(<class 'list'>, {'new key': ['new value']})
```

If you are making a dictionary of lists, you can often write simpler code using defaultdict. In my solution to Exercise 2, which you can get from Here, I make a dictionary that maps from a sorted string of letters to the list of words that can be spelled with those letters. For example, 'opst' maps to the list ['opts', 'post', 'pots', 'spot', 'stop', 'tops'].

> 所以如果你要用列表组成字典的话，你就可以多用默认字典来写出更简洁的代码。你可以在这里下载我给练习2提供的样例代码，其中我建立了一个字典，字典中建立了从一个字母字符串到一个可以由这些字母拼成的单词的映射。例如，『opst』就映射到了列表 ['opts', 'post', 'pots', 'spot', 'stop', 'tops']。

Here's the original code:

> 下面就是原版的代码：

```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        if t not in d:
            d[t] = [word]
        else:
            d[t].append(word)
    return d
```

This can be simplified using setdefault, which you might have used in Exercise 2:

> 用默认集合就可以简化一下，就如你在练习2中用过的那样：

```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d.setdefault(t, []).append(word)
    return d
```

This solution has the drawback that it makes a new list every time, regardless of whether it is needed. For lists, that's no big deal, but if the factory function is complicated, it might be.

> 这个代码有一个不足，就是每次都要建立一个新列表，而不论是否需要创建。对于列表来说，这倒不要紧，不过如果工厂函数比较复杂的话，这就麻烦了。

We can avoid this problem and simplify the code using a defaultdict:

> 这时候咱们就可以用默认字典来避免这个问题并且简化代码：

```
def all_anagrams(filename):
    d = defaultdict(list)
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d[t].append(word)
    return d
```

My solution to Exercise 3, which you can download from Here, uses setdefault in the function has_straightflush. This solution has the drawback of creating a Hand object every time through the loop, whether it is needed or not. As an exercise, rewrite it using a defaultdict.

> 你可以从这里下载我给练习3写的样例代码，该代码中在 has_straightflush函数用的是默认集合。这份代码的不足就在于每次循环都要创建一个 Hand 对象，而不论是否必要。做个练习，用默认字典来该写一下这个程序。

19.8 Named tuples 命名元组

Many simple objects are basically collections of related values. For example, the Point object defined in Chapter 15 contains two numbers, x and y. When you define a class like this, you usually start with an init method and a str method:

> 很多简单的类就是一些相关值的集合。例如在15章中定义的 Point 类中就包含两个数值，x 和 y。当你这样定义一个类的时候，你通常要写一个 init 方法和一个 str 方法：

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
    def __str__(self):
        return '(%g, %g)' % (self.x, self.y)
```

This is a lot of code to convey a small amount of information. Python provides a more concise way to say the same thing:

> 要传达这么小规模的信息却要用这么多代码。Python 提供了一个更简单的方式来做类似的事情：

```
from collections import namedtuple
    Point = namedtuple('Point', ['x', 'y'])
```

The first argument is the name of the class you want to create. The second is a list of the attributes Point objects should have, as strings. The return value from named tuple is a class object:

> 第一个参数是你要写的类的名字。第二个是 Point 对象需要有的属性列表，为字符串。命名元组返回的值是一个类的对象。

```
>>> Point
>>> Point
<class '__main__.Point'>
```

Point automatically provides methods like __init__ and __str__ so you don't have to write them. To create a Point object, you use the Point class as a function:

> Point 会自动提供诸如 __init__ 和 __str__ 之类的方法，所以就不用再去写了。
>
> 要建立一个 Point 对象，你就可以用 Point 类作为一个函数用：

```
>>> p = Point(1, 2)
>>> p = Point(1, 2)
>>> p
>>> p
Point(x=1, y=2)
```

The init method assigns the arguments to attributes using the names you provided. The str method prints a representation of the Point object and its attributes. You can access the elements of the named tuple by name:

> init 方法把参数赋值给按照你设定来命名的属性。 str 方法输出整个 Point 类及其属性的一个字符串表达。
>
> 你可以用名字来读取命名元组中的元素：

```
>>> p.x, p.y
>>> p.x, p.y
(1, 2)
```

But you can also treat a named tuple as a tuple:

> 但你也可以把命名元组当做元组来用：

```
>>> p[0], p[1]
>>> p[0], p[1]
(1, 2)
>>> x, y = p
>>> x, y = p
>>> x, y
>>> x, y
(1, 2)
```

Named tuples provide a quick way to define simple classes. The drawback is that simple classes don't always stay simple. You might decide later that you want to add methods to a named tuple. In that case, you could define a new class that inherits from the named tuple:

> 命名元组提供了定义简单类的快捷方式。缺点就是这些简单的类不能总保持简单的状态。有时候你可能想给一个命名元组添加方法。这时候你就得定义一个新类来继承命名元组：

```
class Pointier(Point):
    # add more methods here
```

Or you could switch to a conventional class definition.

> 或者你可以把命名元组转换成一个常规的类的定义。

19.9 Gathering keyword args 收集关键词参数

In Section 12.4, we saw how to write a function that gathers its arguments into a tuple:

> 在12.4中，我们已经学过了如何写将参数收集到一个元组中的函数：

```
def printall(*args):
    print(args)
```

You can call this function with any number of positional arguments (that is, arguments that don't have keywords):

> 这种函数可以用任意数量的位置参数（就是无关键词的参数）来调用。

```
>>> printall(1, 2.0, '3')
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

But the * operator doesn't gather keyword arguments:

> 但*运算符并不能收集关键词参数：

```
>>> printall(1, 2.0, third='3')
>>> printall(1, 2.0, third='3')
TypeError: printall() got an unexpected keyword argument 'third'
```

To gather keyword arguments, you can use the ** operator:

> 要收集关键词参数，你就可以用**运算符：

```
def printall(*args, **kwargs):
    print(args, kwargs)
```

You can call the keyword gathering parameter anything you want, but kwargs is a common choice. The result is a dictionary that maps keywords to values:

> 你可以用任意名字来命名这里的关键词收集参数，不过通常大家都用kwargs。得到的结果是一个字典，映射了关键词键名与键值：

```
>>> printall(1, 2.0, third='3')
>>> printall(1, 2.0, third='3')
(1, 2.0) {'third': '3'}
```

If you have a dictionary of keywords and values, you can use the scatter operator, ** to call a function:

> 如果你有一个关键词和值组成的字典，你就可以用散射运算符，**来调用一个函数：

```
>>> d = dict(x=1, y=2)
>>> d = dict(x=1, y=2)
>>> Point(**d)
>>> Point(**d)
Point(x=1, y=2)
```

Without the scatter operator, the function would treat d as a single positional argument, so it would assign d to x and complain because there's nothing to assign to y:

> 不用散射运算符的话，函数会把 d 当做一个单独的位置参数，所以就会把 d 赋值股额 x，然后出错，因为没有给 y 赋值：

```
>>> d = dict(x=1, y=2)
>>> d = dict(x=1, y=2)
>>> Point(d)
>>> Point(d)
Traceback (most recent call last):   File "<stdin>", line 1, in <module> TypeError: __new
```

When you are working with functions that have a large number of parameters, it is often useful to create and pass around dictionaries that specify frequently-used options.

> 当你写一些有大量参数的函数的时候，就可以创建和使用一些字典，这样能把各种常用选项弄清。

19.10 Glossary 术语列表

conditional expression: An expression that has one of two values, depending on a condition.

> 条件表达式：一种根据一个条件来从两个值中选一个的表达式。

list comprehension: An expression with a for loop in square brackets that yields a new list.

> 列表推导：一种用表达式， 方括号内有一个for 循环，生成一个新的列表。

generator expression: An expression with a for loop in parentheses that yields a generator object.

> 生成器表达式：一种表达式，圆括号内放一个 for 循环，产生一个生成器对象。

multiset: A mathematical entity that represents a mapping between the elements of a set and the number of times they appear.

> 多重集：一个数学上的概念，表示了一种从集合中元素到出现次数只见的映射关系。

factory: A function, usually passed as a parameter, used to create objects.

> 工厂：一个函数，通常作为参数传递，用来产生对象。

19.11 Exercises 练习

Exercise 1 练习1

The following is a function computes the binomial coefficient recursively.

> 下面的函数是递归地计算二项式系数的。

```
def binomial_coeff(n, k):
    """Compute the binomial coefficient "n choose k".
    n: number of trials
    k: number of successes
    returns: int     """
    if k == 0:
        return 1
    if n == 0:
        return 0
    res = binomial_coeff(n-1, k) + binomial_coeff(n-1, k-1)
    return res
```

Rewrite the body of the function using nested conditional expressions.

> 用网状条件语句来重写一下函数体。

One note: this function is not very efficient because it ends up computing the same values over and over. You could make it more efficient by memoizing (see Section 11.6). But you will find that it's harder to memoize if you write it using conditional expressions.

> 一点提示：这个函数并不是很有效率，因为总是要一遍一遍地计算同样的值。你可以通过存储已有结果（参考11.6）来提高效率。但你会发现如果你用条件表达式实现，就会导致这种记忆更困难。